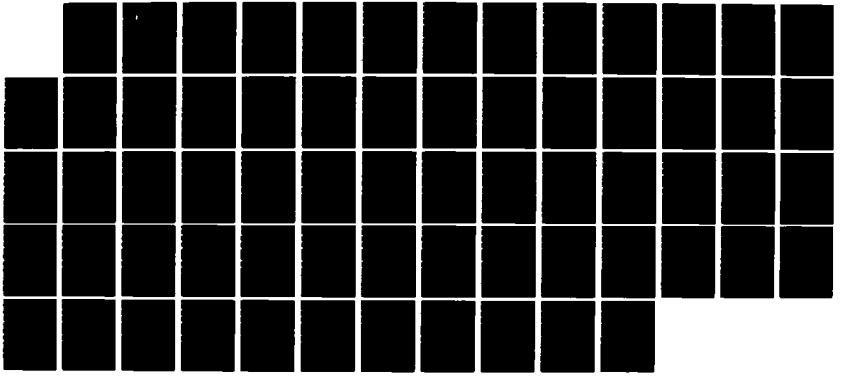
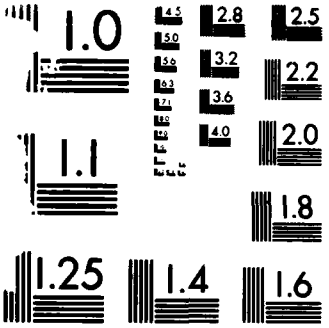
AD-A165 696 FORMAL PROOF OF CORRESPONDENCE BETWEEN THE SPECIFICATION OF A HARDWARE MO... (U) ROYAL SIGNALS AND RADAR ESTBBLISHMENT MALVERN (ENGLAND) C H PYGOTT 1/1

UNCLASSIFIED NOV 85 RSRE-85012 DRIC F/G 9/5 NL





UNLIMITED

8713

(2)

Report No. 85012

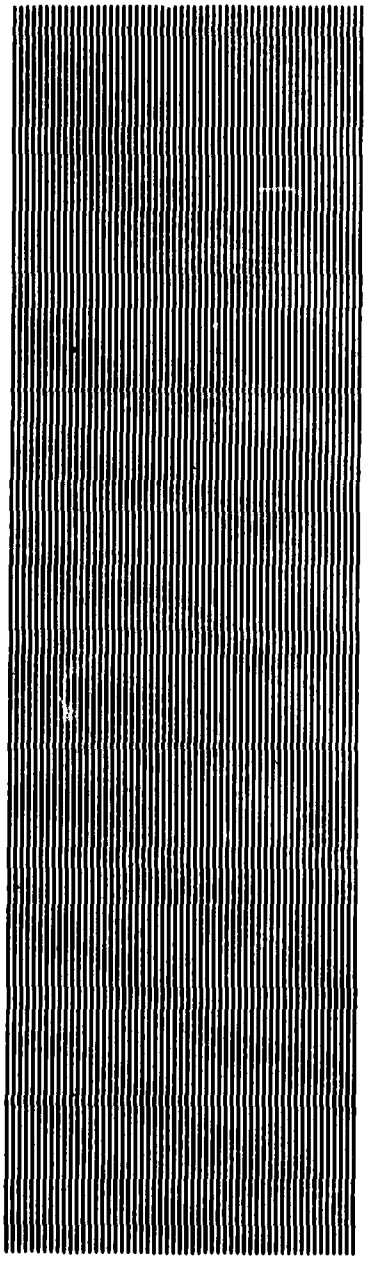
Report No. 85012

ROYAL SIGNALS AND RADAR ESTABLISHMENT, MALVERN

FORMAL PROOF OF CORRESPONDENCE BETWEEN THE SPECIFICATION OF A HARDWARE MODULE AND ITS GATE LEVEL IMPLEMENTATION

Author: C H Pygott

PROCUREMENT EXECUTIVE, MINISTRY OF DEFENCE
RSRE
Malvern, Worcestershire.



November 1985

UNLIMITED

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Report No 85012

FORMAL PROOF OF CORRESPONDENCE BETWEEN THE SPECIFICATION OF A HARDWARE MODULE AND ITS GATE LEVEL IMPLEMENTATION

Author: DR C H PYGOTT

Date: NOVEMBER 1985

The growing use of digital circuits in safety critical environments and the cost of correcting mistakes in large scale integrated circuits, both lead to a requirement for a high level of confidence in the correctness of the design of micro-electronic elements.

This paper describes a novel application of a general hardware description language that enables the implementation of a synchronous circuit to be checked exhaustively against a high level, implementation independent, specification of its functionality (originally written in a formalism such as first order predicate calculus). The technique avoids the cost, in simulation time, usually associated with exhaustive checking.

The method is illustrated by examples written in the design and description language ELLA: no prior knowledge of ELLA is assumed. Included in the annexes to this paper are a library of ELLA functions that provide those facilities required for the validation of circuits, and the translation of specifications written in the first order predicate calculus language LCF-LSM into ELLA.

FORMAL PROOF OF CORRESPONDENCE BETWEEN THE SPECIFICATION OF A HARDWARE MODULE AND ITS GATE LEVEL IMPLEMENTATION

1 Introduction

Digital electronics are being applied, with increasing frequency, to safety critical applications. That is, to those applications where the failure of the digital system will lead to loss of life or serious loss of property (such as aircraft flight control systems and electronic fund transfers). It is obviously vital in such applications that the precise functionality of the system is both known and provable. In general, a digital system will consist of both hardware and software elements. This paper will consider the specification and validation of the hardware elements. The subjects of software specification and validation are covered elsewhere [1].

In order to know the functionality of a piece of hardware, a formal description of its operation is required. A description in English, with a little mathematics, is far too prone to ambiguity and misinterpretation. It is also necessary to be able to show that any implementation of the system does indeed correspond to that specification.

The High Integrity Computing (HIC) section, of the Computing Division of RSRE, have been working for some time on the general problems of the specification and verification of hardware systems. Initial results suggest that precise descriptions of systems can be written, and that verification of implementation with respect to the specification must then proceed in a number of discrete steps. It would appear that correspondence between the 'higher' levels of a description requires the use of theorem provers and other algebraic methods [2]. Whilst these algebraic methods could be applied down to the circuit level, it is believed that this would be undesirable due to the time involved in algebraic analysis. However at the 'lower' levels, that is closer to the circuit description, it appears that a more direct approach is possible, using the existing specification and simulation language, ELLA.

The work to be described developed out of a specific problem. The HIC section has specified a processor, known as VIPER, for use in safety critical systems [3]. The operation of VIPER is specified using first order predicate calculus (specifically Gordon's LCF-LSM [4,5]). By means of careful engineering design, a circuit diagram was produced that claimed to be a realisation of that specification, fabricated as a single UK5000 gate array device [6]. The problem then remained of showing that the circuit and the specification did indeed correspond.

The normal engineering approach to the problem would be to apply test programs to a simulation of the design and the specification, and show that they agree. This approach is adequate for many systems. For highly safety critical devices, the risk of missing some strange, data dependent, faults is too serious to be ignored, and so a more formal testing or analysis method is required. For example, VIPER, which is a 32-bit processor, has some 36 input lines and 200 internal memory elements, meaning that the device is capable of some 2 to the power 236 different state transitions. Even if it were possible to perform a million tests per second, the machine would require a testing time longer than the age of the universe to check them all. In practice, any

such problem would be partitioned to enable smaller sub-units to be tested. However, it is usually impractical to test exhaustively any of these sub-units, and proving that the sub-units can be joined to make a correct whole is often ignored or assumed to be obvious (which it rarely is).

As an alternative to the exhaustive testing of the complete processor, a number of intermediate levels of description have been produced, one of which is a block diagram of the system. Proof that the various levels of description between the top-level specification and the block diagram correspond is achieved by algebraic methods, as described elsewhere [2]. The method used to show that the circuit associated with each block faithfully implements the specification of that block forms the body of this paper. Clearly, if the block level description can be proven to be a correct implementation of the top-level specification, and the circuits for each block can be proven correct, it can be inferred that the circuit correctly implements the top-level specification.

The majority of the work in this paper is expressed in the hardware design and description language ELLA, the main features of which are described in Annex A. However, as has already been stated, the top levels of description for VIPER have been written in LCF-LSM, the main features of which are briefly described in Annex B. In order to allow easy translation from LCF-LSM to ELLA, the types and functions of LCF-LSM have been modelled in ELLA. The ELLA equivalents of the LCF-LSM types and functions are described in Annex C. The text of the ELLA functions is given in Annex E.

In addition to the LCF-LSM library, a library of auxiliary functions has been produced, as described in Annex D, with the text listed in Annex F. These two libraries provide the functions used in the rest of this paper and provide the correct functionality to justify the claims made for the verification process.

It should be noted that the work described here is only concerned with functional correctness. It is believed that there already exists a wide body of knowledge in the CAD field to enable correct physical implementation of a given circuit to be produced and that timing analysis and so on are also well understood. The problems of making hardware fault tolerant or fault detecting (both desirable in safety critical systems) are also essentially separate from correctness.

2 Outline of the 'Method of Intelligent Exhaustion'

In the Introduction it was shown why it is inconceivable that a complete system of any complexity should be tested by exhaustion. Given that the system is broken down into a number of smaller blocks, it is still probable that exhaustively testing these would take too long to be practical. Conversely, it could be said that if the system were broken down into blocks so small that it was possible to test them exhaustively, there would be so many fragments, that very little advantage would have been gained.

A simplistic approach to exhaustive testing ignores the fact that in many cases, not all of the inputs to a system affect the output. For example, consider a four input multiplexer, that is a device with four

data input lines, two control lines and one output. Simplistically, to exhaustively test such a device all 64 (ie 2 to the power 6) input states must be examined to ensure they give the correct output, but one knows from the specification of a multiplexer that when the first data input is selected, the state of the other 3 data inputs should be irrelevant to the output. So, if it were possible to indicate which inputs were irrelevant during a particular test, it would only be necessary to check that when a particular data input was selected, the output had the same value. That is, only two tests are need for each input (input = false, and input = true), or a total of 8 for the whole multiplexer.

For such a testing strategy to work, the dependence of an output on an unexpected input (ie one that was believed to be irrelevant under some particular set of conditions) must be detected and indicated in some way. This leads to a requirement for a redefinition of boolean values to incorporate indeterminate states.

The above example is purely combinatorial. The problem of testing a circuit with memory elements is more complex. In principle, circuits containing memory elements are tested by 'splitting' the circuit at the output of the memory, and describing the function with the current memory state as an explicit input, and the next memory state as an output. That is the memory element is no longer 'buried' in the circuit, and the remaining logic can be tested as though it were combinatorial. This process is illustrated more fully towards the end of this paper.

3 The use of multi-valued boolean logic

The circuits for VIPER have been tested by the above method, with the specification and implementation of each block being described in ELLA [7]. ELLA is a high level logic description language, developed at RSRE, that allows the designer great freedom in the modelling of data types. In this case, this is used to enable a complex boolean type to be created. Some of the facilities of ELLA are outlined in Annex A. The boolean type used in this paper is:-

```
TYPE bool = NEW ( f | t | x | i | q | z | oc | bv).
```

That is a 'bool' has one of eight values. These are:-

- t & f: the normal boolean values of TRUE and FALSE
- x: don't care
- i: indeterminate
- q: the state of a memory element that has not been altered
- z: tri-state high impedance
- oc: open collector output in its "off" state
- bv: a value used in the library in Annex C only

The difference between "x" and "i" is important. "x" is used either when specifying a function to indicate that under some set of conditions the actual value of an output is irrelevant, or is used as the value applied to those inputs of a function that are believed to be irrelevant during some particular test. The value "i" on the other hand is used to indicate any signal whose value is indeterminate or unknowable. For

example, a 2-input NAND gate with "f" and "x" on its inputs should give a "t" output. Whilst, if the inputs are "t" and "x", the output should be "i". This is because, with one input at "t" the output of the NAND gate is the inverse of the other input, but that input is marked as being irrelevant (which is obviously not true in this case) and so the output is indeterminate. "i" can be regarded as an error indicator, which shows that some signal depends upon an input that has been marked as irrelevant or is itself indeterminate.

Returning to the four input multiplexer described in the previous section, its function can be specified as shown below, with the corresponding description of the implementation (see also Fig 1). Note that for the moment the precise definitions of INV NAND3 etc are ignored.

SPECIFICATION

```
FN MUX4S =
   (bool: a b c d select1 select0)
       -> bool:
    CASE (select1, select0) OF
        (f,f): a,
        (f,t): b,
        (t,f): c,
        (t,t): d
     ELSE i
   ESAC.
```

IMPLEMENTATION

```
FN MUX4I =
   (bool: a b c d select1 select0)
       -> bool:
 BEGIN LET select0bar = INV select0,
           select0buf = INV select0bar,
           select1bar = INV select1,
           select1buf = INV select1bar.
 OUTPUT
   NAND4(NAND3(a,select0bar,select1bar)
         NAND3(b,select0buf,select1bar)
         NAND3(c,select0bar,select1buf)
         NAND3(d,select0buf,select1buf)
        )
   END.
```

The specification (left hand side) says that the output equals input 'a' if select0 = "f" and select1 = "f" , input 'b' if select0 = "t" and select1 = "f" etc. Note that if either select0 or select1 has a value other than true or false, the output required cannot be determined and "i" is delivered. It can be seen that the right hand side describes Fig 1, if INV is the functional model of an inverter, and NAND3 and NAND4 are the models of 3 and 4 input NAND gates. It should be noted that the formal parameters listing the inputs and outputs to the functions are identical (which as one is the implementation of the other seems reasonable).

As an aside, it should be stated that one of the reasons for using ELLA in the VIPER work, was that an automated process exists to convert circuit descriptions such as illustrated by MUX4I above into the required input formats for a number of CAD systems. Specifically, ELLA can be used to drive the UK5000 CAD suite.

4 Logical functions applied to multi-valued booleans

Before either the specification of a circuit block or the description of the circuit elements (ie gates) that will be used to implement that block, can be written, the basic logical functions applied to multi-valued logic need to be considered.

One of the most basic logical functions is inversion. For the eight valued 'bool' described earlier, inversion is defined such that the inverse of true is false, the inverse of false is true, and the inverse of any other value is indeterminate. That is expressed in ELLA:-

```
FN NOT = (bool: a) -> bool: CASE a OF t:f, f:t ELSE i ESAC.
```

The logical operation AND is defined for two 'bool's, such that if both are true then the result is true, or if either is false the result is false. All other cases give an indeterminate result. This is expressed in ELLA as (see comment in Annex A.4.3):-

```
FN AND = (bool: a b) -> bool:
  CASE (a,b) OF (t,t):t, (f,bool):f, (bool,f):f ELSE i ESAC.
```

The use of the type name 'bool' as a selector means that any 'bool' value will be accepted. That is "(f,bool):f" means the result is "f" if the first variable has the value "f", independent of the value of the second variable. All the other boolean operations can be defined in the same manner (see Annex E).

5 Specifying and describing the implementation of combinatorial logic

Given the above definitions of the 'bool' type and the logical operations on that type, the designer must now be able to specify the operation of some particular circuit block, and describe the corresponding gate level implementation. For the moment, only combinatorial logic (ie memory free) will be considered. Note that any feedback in the circuit has the same effect as a memory element, and so is not considered in this section.

Whilst writing the specification for a function, the designer is free to use all the facilities of the language. The only restriction is that all combinations of inputs that give indeterminate results, should deliver "i". For example, in the specification of the multiplexer in the previous section, when select0 or select1 have some value other than "t" or "f", the result is "i". This specification is however deficient in one respect, as if select0 and select1 are both "f" the output is 'a', whatever 'a' happens to be. In particular, should 'a' have the value "q", "z", "x", "oc" or "bv", MUX4S will deliver the same value. A function is required that will 'pass' true or false values, but give an indeterminate result for all other inputs. That is:-

```
FN PASS = (bool: a) -> bool: CASE a OF t:t, f:f ELSE i ESAC.
```

The first line of the body of MUX4S should therefore more properly be:-

```
PASS CASE (select0, select1) OF ..... etc
```

The designer may specify that, under some set of conditions, the value of an output is undefined, and so the implementation can deliver any value under those circumstances. For example, an ALU may be defined with 'value' and 'carry' outputs. During addition, both 'value' and 'carry' outputs are significant, and are so defined. However during logical operations, the 'carry' output is meaningless and so can be specified as having the value "x". This implies that no other part of the system will examine the 'carry' output, during a logical operation.

Given a particular implementation technology, such as 74 series TTL or UK5000 gate array, a library of functions is required that model the gates available in that technology. These should be described in terms of the primitive logic operations on the eight valued 'bool's detailed earlier. That is a four input NAND gate, NAND4, could be defined as:-

```
FN NAND4 = (bool: a b c d) -> bool: NOT(a AND b AND c AND d).
```

The circuit can then be described in terms of these gate library functions. As the implementation must be made of functions drawn from this gate library, no other features of the language should be used in the implementation description.

It should be noted that LCF-LSM (and hence the LCF-LSM library in ELLA) can support a number of types other than 'bool'. These include: 'num' being a positive integer value, 'word(n)' being a fixed length row of (n) 'bool's and 'boollist' being a dynamic (unconstrained) row of 'bool's (as described in Annex C).

6 Verification strategy

The verification strategy can be represented diagrammatically as shown in Fig 2. The TESTVECTOR function generates a series of vectors that are chosen to cover all possible input conditions to the required function. The SPEC and CIRCUIT functions are the specification and implementation descriptions, discussed above. The COMPARE function examines the outputs of the specification and implementation, and indicates an error if the two outputs are inconsistent, and the DISPLAY function presents the current vector number and the result of the comparison, together with any other information that may be required, to the user. This additional information would typically be the actual values delivered by SPEC and CIRCUIT in response to a particular test vector. These five functions are unique to a particular problem. The requirements of SPEC and CIRCUIT have already been described and the generation of the others is discussed below:-

6.1 The TESTVECTOR function

The function TESTVECTOR must deliver a series of vectors, each of which is a suitable input for the specification and implementation functions, together with a vector number (used by DISPLAY). The mechanics of making TESTVECTOR deliver the vectors in sequence will not be discussed here, as it would be obvious to anyone using the language and examples can be found in Annexes G and H. However the choice of the vectors is important. They should be chosen by studying the specification function, such that all legal combinations of input states are covered, but at the same time, minimising the total number of test vectors by use of the don't care ("x") input. For normal boolean inputs, this means that the input states "t" and "f" must be covered, but for inputs that are expected to be driven from tri-state sources, the vector must cover the input states "t", "f" and "z". Similarly, if the input is to be driven from an open-collector source, the input states "oc" and "f" must be covered. Note that no vector should include a "q", "i" or "bv" value.

For example, from the specification of the four input multiplexer, MUX4S, it can be seen that the inputs select0 and select1 are always significant, and so each vector must have either "t" or "f" values

for each of these signals. If select0 and select1 both equal "f", then the ouput depends upon the input 'a' only. So 'a' equal to both true and false, with 'b' 'c' and 'd' equal to "x" must be examined. Similarly, with each of the other three values of select0/select1, two more significant vectors can be found. The eight vectors to be produced by TESTVECTOR are therefore (in the order a, b, c, d, select1, select0):-

```
( f, x, x, x, f, f)
( t, x, x, x, f, f)
( x, f, x, x, f, t)
( x, t, x, x, f, t)
( x, x, f, x, t, f)
( x, x, t, x, t, f)
( x, x, x, f, t, t)
( x, x, x, t, t, t)
```

It is trivial to show that these eight vectors cover all 64 possible input states and so are equivalent to exhaustive testing and hence verification.

6.2 The COMPARE function

The COMPARE function requires a new data type 'result' with values "ok", "xxxbadspec" and "xxxwrongxxx" (these rather strange names were chosen to be easily spotted in an output listing). That is:-

```
TYPE result = NEW( ok | xxxbadspec | xxxwrongxxx )
```

To compare a single output from a specification and implementation, a comparison function is required, that takes two 'bool's as inputs, and delivers a 'result'. That is:-

```
FN COMPBOOL = (bool: spec circuit) -> result:
              CASE (spec, circuit) OF
                   ( f,    f): ok,  (t, t): ok,
                   ( q,    q): ok,  (z, z): ok,
                   (oc,   oc): ok,
                   ( x, bool): ok,
                   ( i, bool): xxxbadspec,
                   (bv, bool): xxxbadspec
                ELSE xxxwrongxxx
              ESAC.
```

That is, if the specification requires a "f", "t", "q", "z" or "oc" output, and the implementation delivers the same, then the implementation is consistent with the specification and so COMPBOOL delivers "ok". However, if the implementation delivers some other value, it is inconsistant with the specification and hence COMPBOOL delivers "xxxwrongxxx".

If the specification states that the output doesn't matter ("x"), then COMPBOOL delivers "ok", no matter what value is delivered by the implementation. The only other values that the specification may have are "i" and "bv", but no specification should ever deliver an indeterminate value under the conditions being examined (see 6.1 above) and similarly no function in the library described in the annexes should deliver a value "bv", so if either of these values are found, COMPBOOL delivers the values "xxxbadspec".

The annexes also contain the descriptions of functions for comparing 'word(n)'s and 'num's, COMPWORD(n) and COMPNUM, which are analogous to COMPBOOL, in that they operate upon two objects of the appropriate type and deliver a 'result'.

In general, for circuits which produce more than one output object, the COMPARE function will consist of a number of COMPBOOL (etc) functions, each testing a pair of signals from SPEC and CIRCUIT for consistency. The output from COMPARE is always a single 'result' which should be "ok" if and only if all the output pairs are consistent. The outputs from the separate comparison functions are joined using the function COMPJOIN. This acts on two 'results' and delivers a single 'result'. If both inputs are "ok" the delivered value is "ok". If either input is "xxxbadspec", then the delivered value is "xxxbadspec" otherwise the delivered value is "xxxwrongxxx".

For example, if SPEC and CIRCUIT (called "spec" and "circuit") deliver structures consisting of a 'word4', a 'bool' and a 'num', then COMPARE is as follows:-

```
FN COMPARE = ((word4,bool,num): spec circuit) -> result:
       (COMPWORD4(spec[1], circuit[1])) COMPJOIN
       ( COMPBOOL(spec[2], circuit[2])) COMPJOIN
       (  COMPNUM(spec[3], circuit[3])).
```

6.3) The DISPLAY function

The purpose of the DISPLAY function is to present the results of the comparison to the user. As it is not the intention of this paper to reproduce the whole of the ELLA users guide, the details of running an ELLA simulation will not be described. It is however important to state that the simulator has two modes of operation. These are referred to as "monitor" (MN) and "monitor-changes" (MC).

As was said in the introduction to this section, the minimum information to be presented to the user is the current vector number and the result of the comparison (ie the output of COMPARE). A function DISPLAYRES is provided in Annex D to facilitate this. The mode of DISPLAYRES is:-

```
FN DISPLAYRES = (result: data, num: vectornum, testselect: displaymode)
             -> (num, result, bool):
```

"data" and "vectornum" are the output of COMPARE and the vector number from TESTVECTORS, as already described. "displaymode" is a value of type "testselect" which is used to control the amount of output produced and is usually passed to DISPLAYRES from the simulator (see Annex F). It has the values "all", and "failonly". If "displaymode" has the value "all" and the simulator is run in "monitor" mode, the results of all tests will be printed However, if "displaymode" is given the value "failonly" and the simulator is run in "monitor-changes" mode, only the result of the first test, and any subsequent tests that fail (ie 'result' from COMPARE not "ok") will be printed.

The outputs of DISPLAYRES are (in order), the printable vector number and result of COMPARE (both of which should be delivered from DISPLAY) and a 'bool' which is used to control any other DISPLAY... functions that may be required. That is, if any additional information (such as the responses of SPEC and CIRCUIT to a particular vector) are also to be printed, in order to provide the "print-all" or "print-fail-only" facility described above,

this data (which may be of types 'bool', 'word(n)' or 'num') must be controlled in some manner. For each type there exists a display function (DISPLAYBOOL, DISPLAYWORD(n) and DISPLAYNUM) of the form:-

```
FN DISPLAYTYPE = (type: data, bool: control) -> type:
```

The "control" input to these functions is the third output of DISPLAYRES. So that, if for some circuit, it is required that the values delivered by SPEC and CIRCUIT, of type 'word3', are to be printed together with the result of the comparison and the current test number, the DISPLAY function is:-

```
FN DISPLAY = (testselect: displaymode, result: data, num: vectornum,
              word3: spec circuit
             )
          -> (num,          word3, word3,   result):
             \ vectornumber, spec,  circuit, result\
   BEGIN LET dr = DISPLAYRES(data, vectornum, displaymode).
         OUTPUT (dr[1], DISPLAYWORD3(spec,    dr[3]),
                        DISPLAYWORD3(circuit, dr[3]),
                 dr[2]
                )
     END.
```

A typical output from the above would therefore be:-

```
number/3  t  f  x  t  f  i  ok
```

A complete example of validation of a combinatorial circuit, based on the multiplexer example in given in Annex G.

7 Verifying circuits containing memory elements

When considering circuits which contain memory elements, the above verification strategy needs modification because the output of the function now depends not only upon the inputs, but also upon the current state of the memory elements. The circuit can be regarded as a finite state machine of the form shown in Fig 3a. In order to test such a circuit, it is necessary to control both inputs and the current state of the memory. This is most easily achieved by 'splitting' the circuit at the output of the memory devices and providing the current state of the memory as an explicit input to the function, as shown in Fig 3b. The effect of the complete circuit can be obtained by 'rejoining' the memory output to the current state input, as shown by the dotted line in Fig 3b.

7.1 Modelling memory elements

Before continuing with more details and an example of verifying a circuit containing memory, the model used for a memory element must be considered. It has been said earlier that the method of verification by intelligent exhaustion is only applicable to synchronous circuit, that is to those circuits that have a single distributed clock that is used to clock all latches. In order to illustrate the following remarks about latches, both edge triggered latches (such as 7474 etc) and the latch element in the

UK5000 library will be considered. The UK5000 latch elements are important as that is the technology in which VIPER was designed, and from which the testing scheme evolved.

The simplest use of a latch in a synchronous system is as an ungated temporary store, that is a memory element that samples its input every clock cycle. For the edge triggered latch in Fig 4.a this means that on every rising edge of the common-clock, the D-input (data) is sampled and transfered to the output Q. The UK5000 latch is shown in Fig 4.b has precisely the same function, however as it is known that all latches in UK5000 are controlled from the common-clock, this clock input is not shown.

The second form of memory element is a gated latch, such as shown in Fig 4.c. In this circuit the latch will only sample the input data on the rising edge of common-clock if the gate input is true. By analogy with the simple latch, the UK5000 gated latch is shown in Fig 4.d. This latch model has implications for timing and precisely what the method of verification by intelligent exhaustion can determine about the operation of a circuit. Consider the circuit shown in Fig 4.c, and the timing associated with the clock gating (as shown in Fig 5). As all latches in the circuit (potentially) change state on the rising edge of the common-clock, the logic deriving the gate signal for a particular latch is likely to be in an unstable state for some time after a rising edge, until it finally settles into some stable state. Provided this occurs before the falling edge of the common-clock (as shown in Fig 5.a) this is acceptable. However, if as shown in Fig 5.b, the gate input is unstable after the falling edge of the common-clock, it is possible to incorrectly clock the latch. The ELLA simulator only 'knows' the stable states of the circuit, and so the method of verification by intelligent exhaustion can only prove that "at some low enough frequency, the circuit will operate as specified". It is necessary to use the timing analysis program of the CAD suite used to fabricate the circuit to determine the maximum frequency at which the circuit operates correctly. Note that, although not shown, there is a similar assumption that all data inputs to latches are stable at the next rising edge of common-clock.

7.2 The latch function

There is a further problem that arises when modelling circuits that contain memory elements. In the case of combinatorial logic, already discussed, as well as using the specification and implementation functions (SPEC and CIRCUIT) for verification purposes, either could be used to model the circuit function in a simulation of a more complex system. For example, the design of some system may be *too* complex to verify as a single entity, but may be partitioned into a number of separate blocks, the circuits of which can be verified independently. For formal proof of correctness of the overall system, algebraic methods must then be used, but informally, the designer's confidence in the correctness of the complete system can be increased by simulating it using the SPEC or CIRCUIT functions for the individual blocks. In practice, as SPEC is always simpler than CIRCUIT, SPEC would be used for this purpose. When the circuit contains memory elements this creates a problem as can be illustrated by Fig 3. When the finite state machine is 'split open' for validation, as shown in Fig 3.b, one of the outputs delivered must be the state the memory will have after the next clock cycle, NEXTSTATE, as the specification of a finite state machine must describe the next state it will go into. This should be

compared to the situation when a finite state machine is being simulated, as shown in Fig 3.a. In this case, the memory must be in the correct state for the current clock cycle and must only go into NEXTSTATE during the next clock cycle. That is, during verification, the output of the memory element must be that state it would become on the next cycle, whilst during simulation the memory does not change state until the next cycle.

This leads to a requirement to 'tell' a memory function whether it is being used for validation or simulation. This is done using an object of type "latchmode" which has the values "validate" or "simulate" for the two memory uses. The basic memory element is therefore modelled by a function of mode:-

FN LATCHBOOL = (bool: data gate, latchmode: operation) -> bool:

In verification mode, if "gate" is "f", then the special value "q" is delivered to indicate the contents of the memory do not change. If "gate" is "t", then the function delivers the value of "data", provided "data" is either "t", "f", "q" or "x", otherwise it delivers "i". Any other value of "gate" means that the output is indeterminate and so leads to "i" being delivered.

In simulation mode, if "gate" is "f", then the next value delivered by the function will be the same as the current value. If "gate" is "t", then if "data" is not "q" the value delivered by the function on the next simulation cycle will be "data", modified as described above. However if "gate" is "t" and "data" is "q", the output on the next cycle will be the same as the curent output, that is the latch behaves as if it had been disabled with "gate" equal to "f". Finally, if "gate" has any other value the next value delivered will be "i". It should be noted that the value delivered on the first simulation cycle is "i", indicating that it is impossible to predict which state a memory element will be in after 'switch-on' (see LATCHBOOL in Annex D.2.3).

There is a set of generic functions, analogous to LATCHBOOL, defined in the auxilary library (see Annex D.3) for modelling rows of latches, where the data inputs and outputs are 'word{n}'s. It should be noted that there is no latch primitive for 'num' or 'boollist' data types, as these are essentially unbounded and so cannot be held in latches of finite length.

7.3 An example of specification and verification of a circuit with memory

In order to demonstrate the verification of a circuit containing memory elements, an example of a counter will be used. Informally, the requirement is for a six bit counter, the current state of the counter being "count". The operation of this counter is controlled by two lines, called "func". When "func" is 0, the counter does nothing (ie "count" does not change), when "func" is 1, the counter is loaded with the value on a six bit input bus, called "loadin". When "func" is 2, "count" is incremented and when "func" is 3, "count" is incremented twice. There is a further requirement that there should be a signal coming from the counter to indicate when the contents of the counter are 63. Fig 6.a illustrates this requirement. The formal specification of this requirement, and the whole of the proof of the correctness of the circuit design to meet this requirement, is given in Annex H. The rest of this section will discuss some of the more general points raised by this example.

As was stated in the previous section, the memory element has to be 'split out' of the rest of the circuit before verification can commence. This is illustrated in Fig 6.b. It would therefore be expected that the SPEC and CIRCUIT function would be of the form :-

```
FN SPEC/CIRCUIT = (word6: count loadin, word2: func) -> (word6, bool)
```

That is, taking "count", "loadin" and "func" as inputs, and delivering the next value of "count", "nextcount" and the count equals 63 value, "count63", as outputs. This is correct for CIRCUIT, but as was stated earlier, the SPEC function may be used for either verification or simulation. This means that a 'latchmode' input is required for SPEC, to indicate its current use. This could be added as a forth input parameter, as:-

```
(word6: count loadin, word2: func, latchmode: mode)
```

but this makes later functions untidy. It is neater to keep the three inputs that correspond to the inputs of CIRCUIT as a single input structure, and to add the 'latchmode' input as a second parameter, as:-

```
((word6,word6,word2): signals, latchmode: mode)
```

The members of "signals" must of course be then named within the SPEC function. That is "LET count = signals[1]." etc (as shown in Annex H.2).

Having produced the SPEC and CIRCUIT functions, consideration must be given to the test vectors required to show correspondence between the two. Firstly, it should be noted that the system as shown in Fig 6.b has 14 inputs, and so would require 16384 tests to perform exhaustive checking by a simplistic method. It will be shown that using intelligent exhaustion the same effect can be achieved with just 148 tests.

It is obvious from the specification that "func" provides a major control over the value of "nextcount", so testing for this output will be considered for the four possible legal values of "func".

When "func" is 0, the next state of count is unchanged, as indicated by "nextcount" being "[6]q". It can be seen that this value is independent of both the current value of "count" and "loadin". So a single vector with "count" and "loadin" both equal to "[6]x" is sufficient to check this condition. Whilst it would not have been incorrect to specify "nextcount" as "count" in this case, proof of correspondence would then have required 64 vectors (ie all legal values of "count").

When "func" is 1, the value of "nextcount" is independent of the current state of "count" and so the "count" input will have the value "[6]x". As the next state of "count" is defined as "loadin" it would be possible to show for each of the 64 legal values of "loadin" that "nextcount" was correct, however it is more efficient to show that each bit of the "nextcount" depends only upon the equivalent bit of "loadin". That is the first bit is tested with "loadin" equal to "(f,x,x,x,x,x)" and "(t,x,x,x,x,x)" etc. Therefore just twelve vectors are needed.

When "func" is 2 or 3, the value of "nextcount" depends upon arithmetic operations performed on the current value of "count", but is independent of "loadin". Therefore in each of these cases the 64 possible legal values of "count" must be tested, with "loadin" equal to "[6]x". That is 128 vectors are needed.

Finally, the "count63" output must be shown to be correct when the current value of count is 63 (ie "[6]t") and when any bit is "f", "loadin" being "[6]x". That is seven vectors are required (see comments in section H.2).

That is, the total number of vectors required is just 148. Annex H details the formal specification, the circuit to implement this specification and the functions required to perform the above verification.

It is also possible to use the SPEC function as a model of the counter in a larger simulation. This requires the "nextcount" output of SPEC 'joining' back to the "count" input, as Fig 6.a. This is achieved by the following function:-

```
FN COUNTER = (word6: loadin, word2: func) -> (word6,bool):
   BEGIN MAKE SPEC: specification.
         JOIN ((specification[1],loadin,func), simulate) -> specification.
         OUTPUT specification
     END.
```

8 Application to the verification of the VIPER microprocessor

In the above examples, the technique described has led to a reduction in the number of tests required for verification. It is recognised that these were simple examples that could have been tested exhaustively anyway. However, as stated in the Introduction, the method described was developed to solve a particular practical problem. The design of the VIPER processor naturally divided into nine major blocks, and this partitioning existed before the method of testing was developed. Of these, only the ALU required further sub-division. This was divided into eight identical 4-bit ALU slices plus various circuits to 'glue' the whole ALU together. The total design was implemented in some 4000 gates (each gate being the equivalent of a two input NAND or NOR).

One of the circuit blocks was an instruction decoder, having 18 inputs and 26 outputs and consisting of some 500 gates of random logic. Compared to the quarter of a million possible input states that would be necessary for conventional exhaustive testing, this circuit was verified using 1200 tests produced in the manner described above. Furthermore, this verification found a design error that would have been virtually impossible to find by simulation. If VIPER had been reset during the execution of a call instruction, one particular flag bit would have failed to clear. The chances of a simulation attempting to perform a reset during that particular instruction, whilst that particular flag bit was true, is so small that the error would very likely have remained undetected. It is just this kind of improbable, data dependent, fault that is unacceptable in a safety critical system.

The whole of VIPER required some 6000 tests to verify the various component parts, and by simulating the whole chip using the specification of the parts, rather than the implementations, it has been possible to run some very large hardware test programs, that would have been impractical on a gate level simulation.

9 Conclusion

The application of the technique described in this paper to VIPER has shown that it is possible to specify the functionality of comparatively large circuit elements, and to show that a gate level implementation of these elements is consistent with that specification. The size of circuit element that can be treated in this way depends upon its functionality. For example n-bit arithmetic and parity operations require 2 to the power n tests, as all input bits are significant in determining the result. However, this method still allows the system to be partitioned into few large blocks, the overall effect of which can be determined algebraically.

Work to date has concentrated on proving the validity of the method of proof and its application to VIPER. However, a number of possible future lines of research are clear. The main aim of this future work would be to incorporate the above methodology into a 'user friendly' environment. The first such improvement would be to provide a tool to check that the test vectors defined to verify a particular circuit did provide complete coverage. An extension of this would be to automatically generate the test vectors from the SPEC function. Finally, as it seems likely that the algebraic manipulations will be performed in the language LCF-LSM rather than ELLA, a translator from ELLA to LCF-LSM (or more likely LCF_LSM to ELLA) would seem desirable.

Whilst the motivation behind this work has been the necessity to prove the correctness of circuits to be used in safety critical environments, it is believed that with the increasing complexity of VLSI circuits, it will be necessary to employ more formal design verification techniques, such as described in this paper to a far wider range of products for purely commercial reasons.

Acknowledgements

The author wishes to thank Dr W J Cullyer, Dr R J W Kershaw and Mr J D Morison for their help and advice during the development of the ideas expressed in this paper.

References

[1] CULLYER W.J.
"Software design methods"
Proc. Design & Advanced Concepts of Avionics/Weapons Systems Integration Colloquium.
Royal Aeronautical Society, London 1984

[2] CULLYER W.J., PYGOTT C.H.
"Hardware proof using LCF-LSM and ELLA"
RSRE memo 3832.

[3] KERSHAW R.J.W.
"Safe control systems and the VIPER microprocessor"
RSRE memo 3805.

[4] GORDON M.J., MILNER R.A., WADSWORTH C.P.
"Edinburgh LCF"
Lecture Notes in Computer Science, 1979, Springer-Verlag

[5] GORDON M.J.
"LCF-LSM"
University of Cambridge Computing Laboratory,
Technical Report 41

[6] GRIERSON J.R., COSGROVE B., DANIEL R., HALLIWELL R.E., KIRK I.H., KNIGHT J.C., MCLEAN J.A., MCGRAIL J.M., NEWTON C.O.
"The UK5000, Successful collaborative development of an integrated design system for a 5000 gate CMOS array with built-in test"
PROC. ACM IEEE Design Automation Conference,
Miami Beach, June 1983

[7] MORISON J.D., PEELING N.E., THORP T.L.
"ELLA: Hardware description or specification?"
PROC. IEEE International Conference CAD-84,
Santa Clara Nov12-15 1984

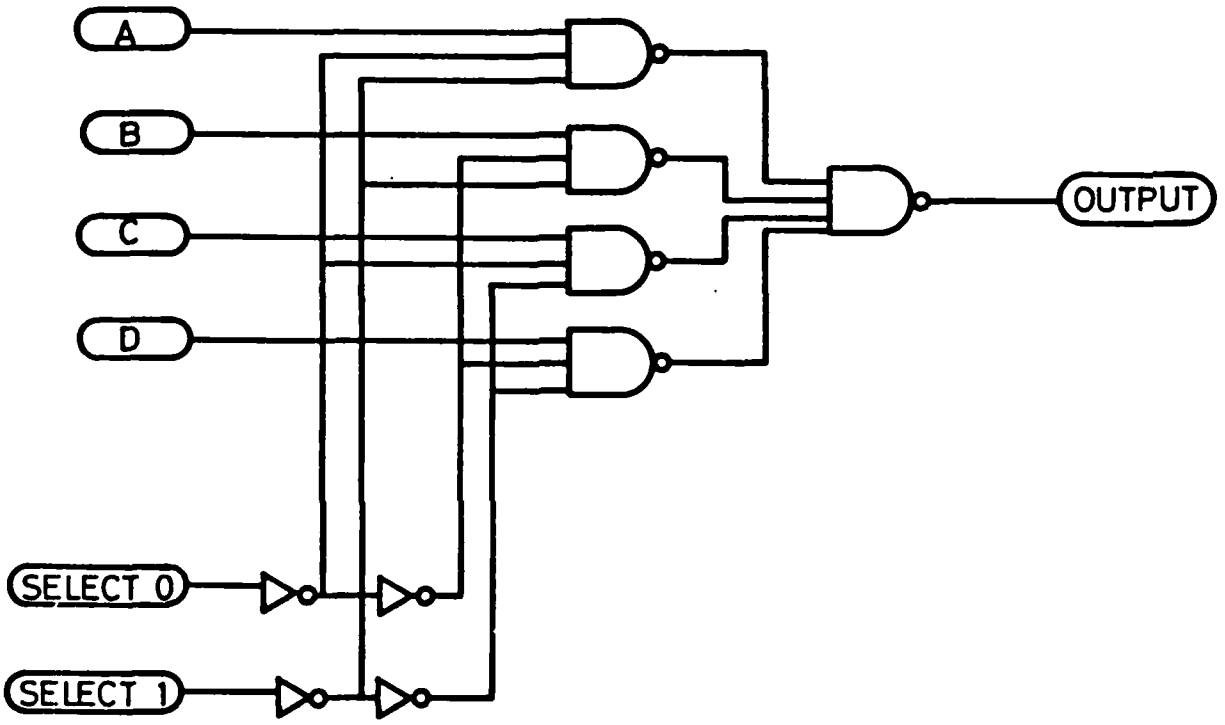


FIG.1 IMPLEMENTATION OF A FOUR INPUT MULTIPLEXER

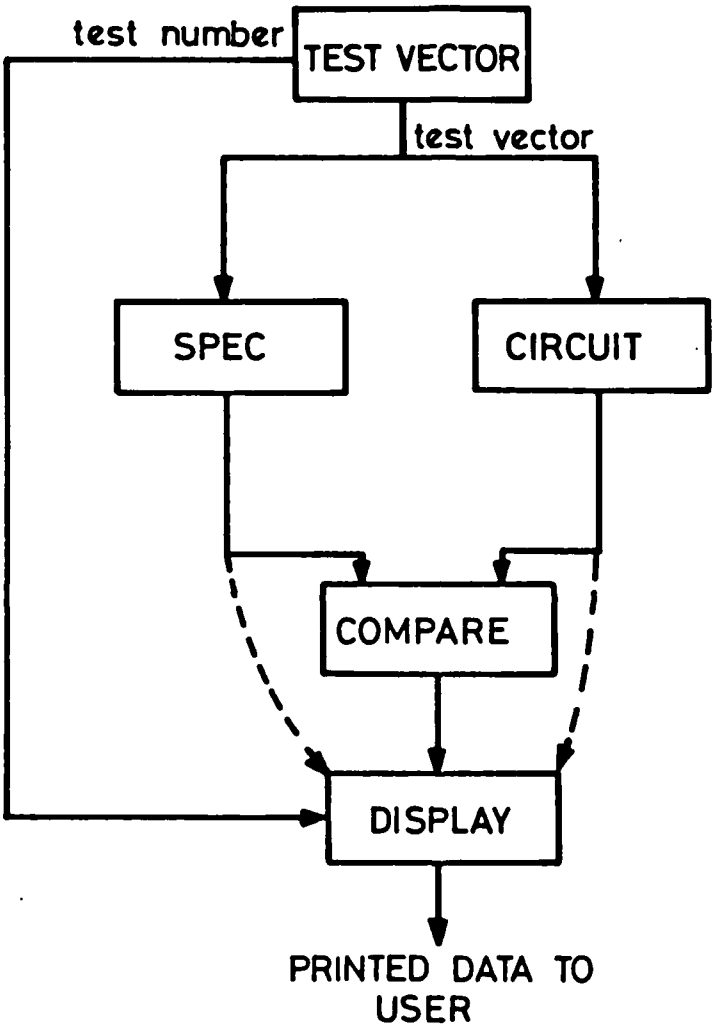


FIG. 2 DIAGRAMATIC VERIFICATION STRATEGY

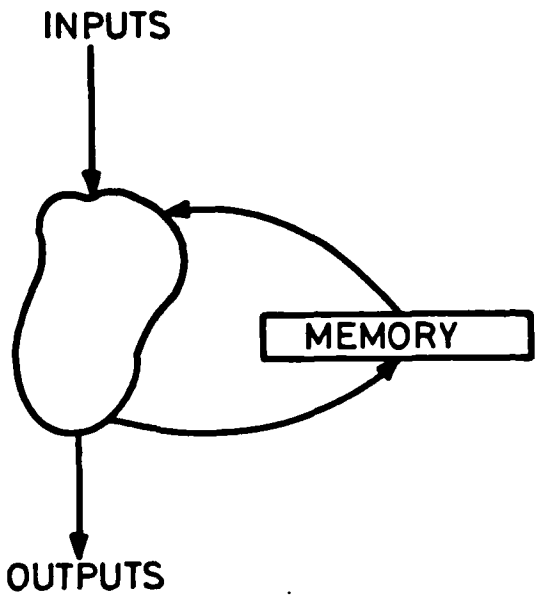


FIG.3a FINITE STATE MACHINE

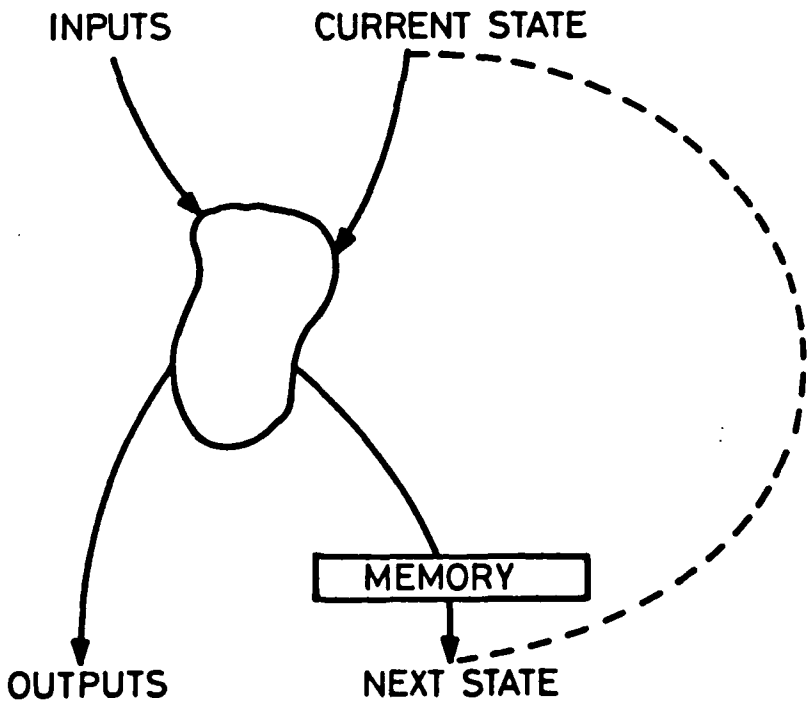


FIG.3b FINITE STATE MACHINE FOR VALIDATION

FIG.3 VERIFYING A CIRCUIT WITH MEMORY

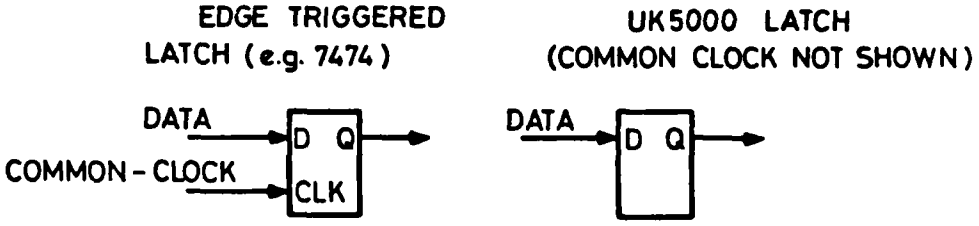


FIG. 4a FIG. 4b

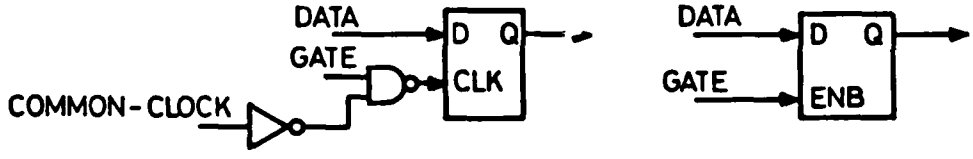


FIG. 4c FIG. 4d

FIG. 4 EXAMPLES OF LATCH ELEMENTS

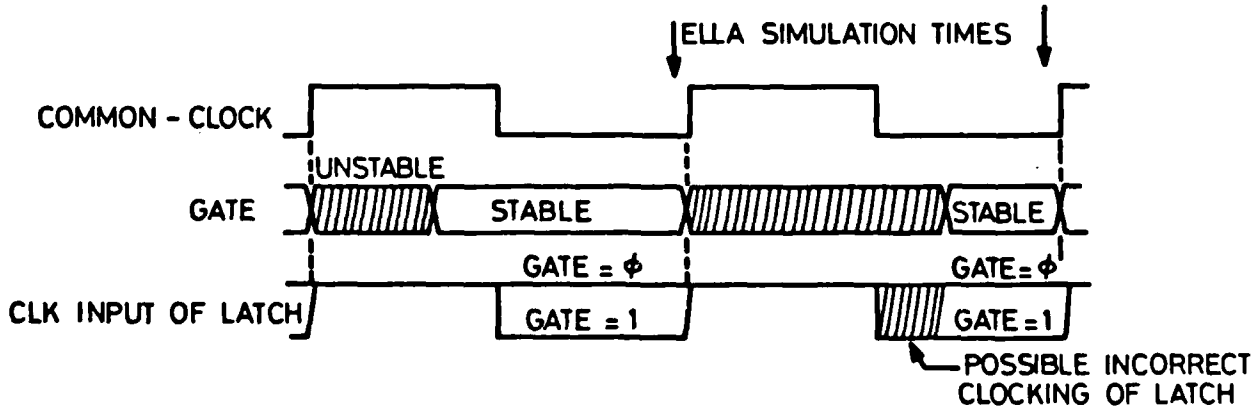
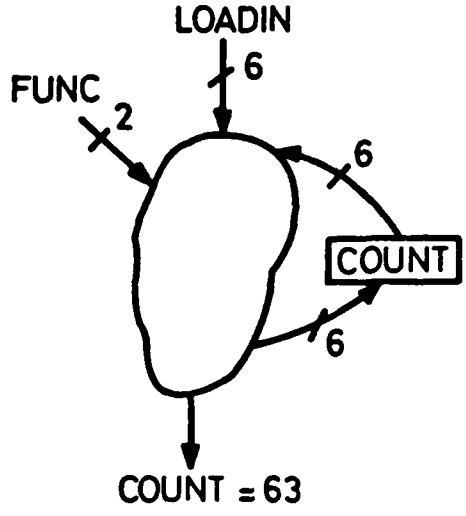


FIG. 5a FIG. 5b

FIG. 5 GATED EDGE TRIGGERED LATCH TIMING (SEE FIG. 4c)



FUNC	NEXT VALUE OF COUNT
ϕ	NO CHANGE
1	LOADIN
2	COUNT + 1
3	COUNT + 2

FIG. 6a REQUIREMENT FOR COUNTER CIRCUIT

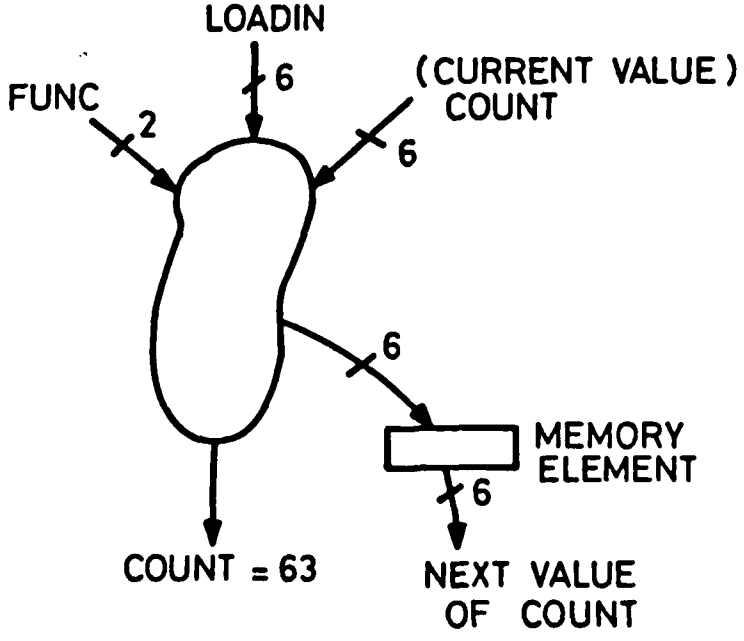


FIG. 6b COUNTER CIRCUIT 'SPLIT OPEN' FOR VALIDATION

FIG. 6 A COUNTER

Annex A: ELLA: A brief introduction to its syntax and semantics

This annex will outline those features of ELLA used in this paper. It contains more detail than could be included in the body of the paper, but is not intended to be a complete description of the language.

All ELLA programs consist of type declarations and functions only. There are no global variables or constants (other than integer constants) in ELLA. Also, ELLA has no in-built data types, so all data types to be used must be declared explicitly.

A.1 Primitive data types

Two sorts of primitive data types can be declared; enumeration and integer types. The declaration of an enumeration type consists of the type name, such as "bool", and a list of the values it may have, such as "t" "f" "x" and "z". That is:-

```
TYPE bool = NEW( t | f | x | z ).
```

Note the order of the values in the declaration is irrelevant. Integer types consist of the type name, such as "countint", a prefix name, such as "count" and the range of values that integers of this type may have, such as 0 to 10. That is:-

```
TYPE countint = NEW count/(0..10).
```

Variables of type "countint" can have the values "count/0" "count/1" etc to "count/10". The prefix "count" distinguishes integers of type "countint" from integers of any other type (which would have a different prefix).

Integer constants (for the size of arrays etc) can be defined as:-

```
INT arraysize = 24.
```

A.2 Compound data types

Compound data types are collections of primitive data types or other compound data types. They are of two forms; rows and structures. A row is a collection of identical data types. For example, "(t,f,t)" is a row of three "bool"s, the type of this object can be expressed as either "(bool,bool,bool)", or "[3]bool". Structures are collections of different data types, such as "(t,count/0,(t,f))". The type of this object is "(bool,countint,[2]bool)".

Both rows and structures are indexed in the same way. If the above example of a structure was called "struct", then "struct[1]" would be the first element of the structure, that is the "bool" with value "t". Similarly, "struct[2]" is the "countint" with value "count/0", and "struct[3]" is the "[2]bool" with value "(t,f)". This row can be indexed in the same manner, such that "struct[3][1]" is a "bool" with value "t", and "struct[3][2]" is a "bool" with value "f". Groups of row elements can be selected by "[index..index]". For example, "a[2..4]" is a row of three objects selected from row "a".

A name can be given to a compound type in the following manner:-

```
TYPE struct = (bool, countint, [2]bool).
TYPE word16 = [16]bool.
```

A.3 Functions

Functions in ELLA are similar to mathematical functions, in that they deliver a value and can only operate upon those values passed to the function when it is used. That is, there are no global variables.

A function consists of a heading and a body. The heading describes the types of the objects that the function will operate upon, together with their local names (ie the names by which the parameters are known within the function body) and the type of the object delivered by the function. For example, consider a function called "TIMEOUTS", which is to operate on two objects of type "bool", and one object of type "[3]bool" and is to deliver a structure with "bool" and "[3]bool" elements. The names, within this function, of the objects to be operated upon are "reset" "inc" and "current". That is the function heading is:-

```
FN TIMEOUTS = (bool: reset inc, [3]bool: current) -> (bool, [3]bool):
```

A function body consists of either a single 'expression' (qv), or "BEGIN" followed by a number of 'statement's (qv) "OUTPUT" followed by an expression and "END.". The value delivered by the function in the first case is the value of the expression, and in the second is the value of the expression between "OUTPUT" and "END.". The type of the delivered value must be the same as that indicated in the function heading.

A.4 Expressions

There are four types of ELLA expression. All of them have the property that they deliver a value. They are; simple, function calls, CASE and ARITH.

A.4.1 Simple expressions

These are structures composed of explicit data values, names local to the function containing the expression, or other expressions. Explicit values are those values declared as being of a particular data type, such "t" or "count/4" in section A.1. Local names are the values associated with the parameters named in the function heading or the value associated with a named expression (see LET, section A.5.1). Note that a single value can also be a simple expression. For example:-

```
t     inc     (reset, inc, (t,t,t))     are simple expressions.
```

A.4.2 Function calls

An expression can be the result delivered by applying a function to a particular set of values. The values operated upon can be any sort of expression including simple expressions (ie explicit values or local names). Given a function called OR that operates on two "bool"s and delivers a "bool", the OR of "a" and "b" (where "a" and "b" are local named values of type "bool") is given by "OR(a,b)".

There are two exceptions to this rule. If the function has a single parameter, the brackets are not needed. So "NOT(a)" can be written as "NOT a". If the function has two parameters, it can be placed between the values it is to operate upon. So "OR(a,b)" can be written as "a OR b". Similarly "a OR b OR c OR d" is the same as "OR(OR(OR(a,b) ,c) ,d)".

A.4.3 CASE expressions

The structure of a CASE expression is:-

CASE expression OF {value: expression} ELSE expression ESAC

Where {...} means repeated any number of times.

The first expression is evaluated and the resulting value is compared with the 'value' component of each 'value: expression' pair. If any of these match, the value delivered by the CASE expression is the value of the associated 'expression' component. If none of the 'values' are equal to the evaluated value, the value of the expression between ELSE and ESAC is delivered. If it is known that the 'value: expression' pairs cover all possible values the "ELSE expression" term may be omitted.

The ELLA compiler checks that the various limbs of the CASE are non-overlapping, that is given a particular value for the expression between CASE and OF only one of the 'value: expression' pairs should be selectable. If any of the pairs do overlap, this will normally be indicated by a compilation error. However by using ELSEOF, overlapping pairs can be allowed. The function AND in section 4 of the body of the paper is illegal in this respect (but textually simpler than the legal version). It is stated as:-

```
FN AND = (bool: a b) -> bool:
   CASE (a,b) OF (t,t): t, (bool,f): f, (f,bool): f ELSE i ESAC.
```

If 'a' and 'b' are both 'f', then either the second or third limbs could be selected (although they would both give the same result), so this would be rejected by the compiler. The correct function is:-

```
FN AND = (bool: a b) -> bool:
   CASE (a,b) OF
        (t,t): t,  (bool,f): f  ELSEOF (f,bool): f
     ELSE i
   ESAC.
```

A.4.4 ARITH expressions

If a function is required to perform arithmetic operations on integer types, and deliver an integer type result, the body of the function may be a single ARITH expression. The required arithmetic operation may be expressed in an ALGOL like manner using the operators "+", "-", "*", "%" etc. "%" is used for integer divide ("/" having been used in the representation of ELLA integers). The operators "SL" and "SR" also exist, meaning shift left and shift right. Condition clauses may be formed using an IF..THEN..ELSE..FI construct.

A.5 Statements

There are three types of statement that may form the body of a function. Note that when a number of expressions of the same type is required, the statement indicator (LET, MAKE or JOIN) is only required once. So that, "LET a = t, b = f." is the same as "LET a = t. LET b = f.".

As well as LET, MAKE and JOIN, a function body may also contain the definition of a local function, using the format described in A.3.

A.5.1 LET

The LET statement allows a name to be associated with the value of an expression. So that "LET aorb = a OR b." means that "aorb" is now a recognised local name associated with the value of the expression "a OR b".

A.5.2 MAKE and JOIN

In all the above examples, local names must have been declared as the parameter of a function or a LET statement, before they could be used in an expression. Without some means of overcoming this restriction it would be impossible to model circuits with feedback (and hence memory). Consider a pair of cross coupled NAND gates (as shown in Fig A.1). The description of this circuit as:-

```
FN RSLATCH = (bool: a b) -> bool:
   BEGIN LET nand1 = NAND(a, nand2),
             nand2 = NAND(b, nand1).
         OUTPUT nand1
     END.
```

is illegal, as "nand2" is used in an expression before it is declared. MAKE allows a name to be associated with the output of a particular call of a function before the inputs to that function are available. JOIN allows the required inputs to a function named by MAKE to be connected after they have been declared. Hence, a legal version of the same function would be:-

```
FN RSLATCH = (bool: a b) -> bool:
   BEGIN MAKE NAND: nand2.
         LET nand1 = NAND(a, nand2).
         JOIN (b, nand1) -> nand2.
         OUTPUT nand1
     END.
```

A.6 DELAY and ELLA's model of time.

The ELLA simulator, used to animate ELLA text, calculates the values of all functions and variables at a series of discrete time steps. All the operations described so far are evaluated in zero time (as regarded by the simulator). In order that some circuits with feedback can be modelled, it is necessary to provide delays.

A special expression, "DELAY", exists which is used to create functions which will act to delay a signal for a number of 'simulation states'. It is used as:-

```
FN DELAYANY = (anytype) -> anytype: DELAY(value, integer).
```

This defines a function, DELAYANY, that delays a signal of type 'anytype' for 'integer' clock ticks. 'value' gives the initial value of DELAYANY's output.

For example, if NOT is a function that provides boolean inversion, and DELAYBOOL is a function that delays a 'bool' by one simulation cycle. Then consider the following two functions:-

```
FN F1 = (bool: dummy) -> bool:          (see Fig A.2)
   BEGIN MAKE NOT: invert.
         JOIN invert -> invert.
         OUTPUT invert
     END.
```

```
FN F2 = (bool: dummy) -> bool:          (see Fig A.3)
   BEGIN MAKE NOT: invert.
         JOIN DELAYBOOL invert -> invert.
         OUTPUT invert
     END.
```

Both functions describe circuits consisting of single inverter with the output connected back to the input. In F1, there is no delay involved, so ELLA simulator will be unable to calculate the value to be delivered (and will deliver "?"), as the output is defined as being the inverse of the output (which is clearly paradoxical). In the case of F2 however, with a single state delay in the feedback path, the output at time "T" is defined as the inverse of the output at time "T-1", which is calculable. F2 in fact models an oscillator.

Note that in both functions the input parameter 'dummy' is not used. However all ELLA functions must have at least one input parameter as functions of type "VOID -> bool" (to use ALGOL68 notation) are not allowed.

A.7 MACRO's

MACRO's allow the definition of groups of similar functions to be written as a single parameterised function and then characterised with explicit parameters when required.

For example, consider a function that shortens a row of 'bool's by one element. For a general row of length "n", this function can be written as:-

```
MAC SHORTEN(INT n) = ([n]bool: a) -> [n-1]bool: a[1..(n-1)].
```

Specific functions for given values of "n" can then be produced as:-

```
FN SHORTEN4 = ([4]bool: a) -> [3]bool: SHORTEN(4) a.
```

An "IF ... THEN ... ELSE ... FI" construct is allowed within a MACRO to produce different functions for different input parameters.

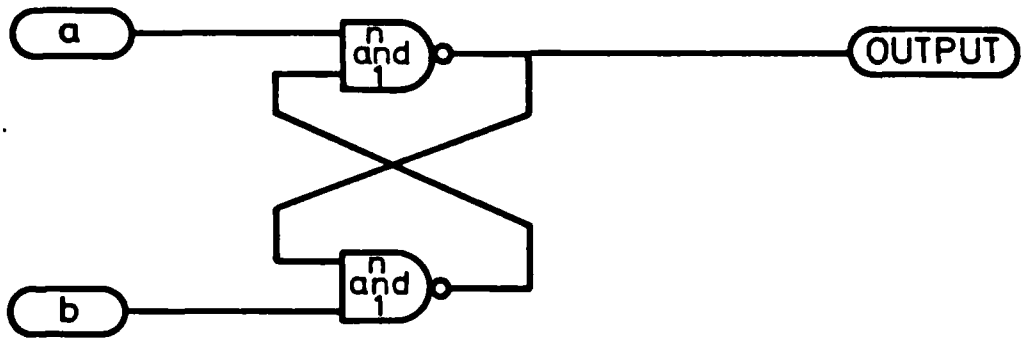


FIG. A.1 AN RS LATCH

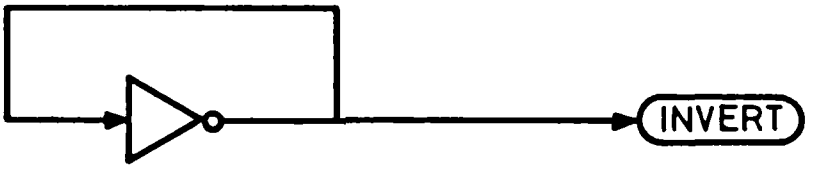


FIG. A.2 AN UNSTABLE FUNCTION (F1 SECTION A.6)

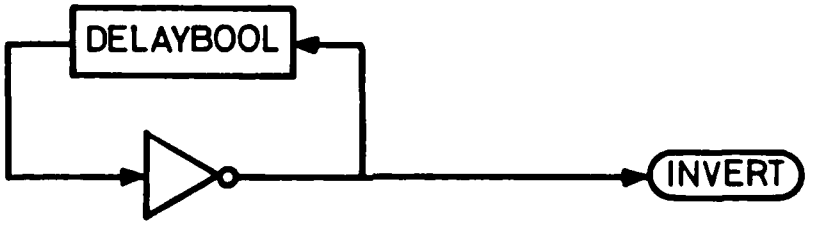


FIG. A.3 AN OSCILLATOR FUNCTION (F2 SECTION A.6)

Annex B: An outline of LCF-LSM

The material in this annex is a very brief digest of that presented by Gordon in References 4 & 5 but contains enough detail to enable comparison to be made with the library of functions described in Annex C.

The form of LCF-LSM in use has the following built-in types:-

'bool' with values T and F

'num' with values 0, 1, 2 etc (without limit)

'word(n)' being a collection of (n) binary objects (each object having the value 0 or 1). A word(n) corresponds to an (n)-bit computer word. The value of the 'word4' representing the decimal value 11, is written as #1011.

'? list' A list of objects of any type. For example, 'bool list' is a list of 'bool's. It should be noticed that a list is unconstrained and may contain any number of elements (numbered from 0).

LCF-LSM supports a number of operations on these types, only the function names are given here, with a brief statement of their effect:-

=	test of equality between objects of the same type	?#? -> bool
+	numerical addition	num#num -> num
-	numerical subtraction	num#num -> num
*	numerical multiplication	num#num -> num
/	numerical division	num#num -> num
NOT	boolean inversion	bool -> bool
OR	boolean disjunction	bool#bool -> bool
AND	boolean conjunction	bool#bool -> bool
XOR	boolean exclusive OR	bool#bool -> bool
CONS	list constructor (any type)	?#? list -> ? list
HD	head of list	? list -> ?
TL	tail of list	? list -> ? list
NULL	check for empty list	? list -> bool
EL	deliver selected element from list	num#? list -> ?
SEG	deliver selected set of elements from list	(num#num)#? list -> ? list
V	convert a bool list to the equivalent number	bool list -> num

Generic functions parameterised for values of (n)

WORD(n)	convert a number to the equivalent word(n)	num -> word(n)
BITS(n)	convert a word(n) to the equivalent bool list	word(n) -> bool list
VAL(n)	convert a word(n) to the equivalent number	word(n) -> num
NOT(n)	invert each bit of a word(n)	word(n) -> word(n)
AND(n)	bitwise conjunction	word(n)#word(n) -> word(n)
OR(n)	bitwise disjunction	word(n)#word(n) -> word(n)

Annex C: Description of the LCF-LSM library expressed in ELLA

C.1 Supported types

C.1.1 bool

This is the ELLA enumeration type, equivalent to the LCF-LSM type "bool". It has eight values:-

t: True) - These are the normal boolean values, and are the
f: False) only values recognised by LCF-LSM

i: Illegal or indeterminate, used either to show that some illegal action was attempted, or that the result of some function is indeterminate, such as AND(t,x)

x: Unknown, used is an input to a function under test to indicate that that input should be irrelevant to the output, or as the output of a specification function to show that a particular output is not being used under some set of circumstances and that the implementation could deliver any value

q: Unchanged, used as the output of a latch when the enable input has not been selected (see latch functions, D.2.3). That is it represents a stored boolean value that has not altered.

bv: A value used inside the library functions only (ie should never be delivered). It is used as the value of an "empty" element in a boollist.

z: High impedance, used in the description of tri-state logic.

oc: Open collector, used in the description of open collector logic, for an output transistor in its "off" state. An open collector output transistor in its "on" state is represented by "f".

C.1.2 num

This is an ELLA integer type equivalent to the LCF-LSM "num" type. It should be noted that whilst the range of LCF-LSM 'num's is the entire range of positive integers (ie 0 to infinity), ELLA's number range is limited. The ELLA 'num' type is defined with a range of legal values equal to: 0 to 262143 (2**18 - 1). Two other values are defined to indicate particular conditions:-

number/0 to number/262143: The range of positive integers used as legal values.

number/262144: Called "illegalnum", this is the number used to indicate an illegal or indeterminate 'num' (cf 'i' in the definition of bool).

number/262145: Called "unknownnum", used is an input to a function under test to indicate that that input should be irrelevant to the output, or as the output of a specification function to show that a particular output is not being used under some set of circumstances and that the implemenation could deliver any value (cf 'x' in the definition of bool)

Note the "integer type prefix" in ELLA is "number", and that there is no analogue for the 'bool' value 'q'. This is because 'num's are essentially unbounded objects and so cannot be stored in a finite sized latch.

C.1.3 boollist (& maxlist)

This is the ELLA equivalent of the LCF-LSM "boollist" type. It should be noted that the LCF-LSM 'boollist' is unconstrained (ie may contain 0 to infinite members), whilst the ELLA boollist is represented by a finite row of 'bool's. The first declaration in the library, "INT maxlist = ????", sets the length of this row, and hence the maximum length of 'boollist' that ELLA can support. This value must lie between 19 and 256 (inclusive).

When a boollist is constructed from a row of bools, the significant values are held in the least significant end of the boollist, whilst the rest of the boollist is filled with 'bv's. That is if a boollist is represented as a row of 32 'bool's, and comprises 10 significant values (ie 't', 'f', 'i', 'x', 'q', 'z' or 'oc'), in ELLA terms, the boollist is represented with elements 1 to 10 being the significant values, whilst elements 11 to 32 contain 'bv'. Notice however that LCF-LSM numbers the boollist from element 0, so if EL or SEG are used to examine the contents of the boollist, elements 0 to 9 are the significant values, whilst 10 to 31 are 'bv'.

Three special values of boollist should be noted. These are an empty list, represented by "[maxlist]bv", a corrupted list, ie one that has been corrupted by an illegal or ill-formed operation (represented as "[maxlist]i") and an unknown/irrelevant list (represented by "[maxlist]x"). Note that there is no representation of an unaltered stored list, as boollists are essentially dynamic objects and so cannot be held in latches of finite length.

C.1.4 word(n)

This is the ELLA equivalent of the LCF-LSM "word(n)" types, representing a fixed length row of 'bool's. When wishing to use a word(n) type, the user should make a declaration of the form:-

"TYPE word16 = [16]bool." for (n) = 16

The value of (n) should not exceed 18, unless only logical operations are to be performed on the words, as larger 'word(n)'s cannot be converted into 'num's. Allowable operations are:- BITS(n), NOT(n), AND(n) and OR(n).

Illegal, unchanged stored, unknown, high impedance and open collector 'word(n)'s are represented by "[n]i", "[n]q", "[n]x", "[n]z" and "[n]oc" respectively.

C.2 Supported simple functions

C.2.1 Boolean operations

NOT: (bool: a) -> bool
This is the inversion operation. If 'a' is 't' or 'f' the result is 'f' or 't' as would be expected. If 'a' has any other value the result is 'i'.

AND: (bool: a b) -> bool
This is the 'and' operation. It is defined so that "AND(t,t)" is 't', whilst "AND(anything ,f)" (or AND(f, anything)) is 'f', as would be expected. Under all other circumstances the result is 'i'. Note specifically that "AND(t,x)" and "AND(t,i)" are 'i'.

OR: (bool: a b) -> bool
This is the 'or' operation. It is defined in an analoguous way to AND above.

EQUIV: (bool: a b) -> bool
This is the equivalence operation. If 'a' and 'b' are both either 't' or 'f', then if 'a' = 'b' the result is 't', if 'a' is different to 'b' the result is 'f'. In all other cases the result is indeterminate ('i').

XOR: (bool: a b) -> bool
This is the "exclusive or" operation, and is equivalent to NOT(EQUIV(a, b)).

PASS: (bool: a) -> bool
If 'a' is either 't' or 'f' then the result is 'a', otherwise the result is 'i'. That is this function ensures that the result of a calculation is either 't', 'f' or 'i', and prevents the other special values of 'bool' from appearing in the output of a function.

C.2.2 Numerical operations

EQUAL: (num: a b) -> bool
If 'a' and 'b' are both legal numbers (0 to 262143), then if 'a' equals 'b' the result is 't', or if 'a' is not equal to 'b' is result is 'f'. If either input is not a legal number then the result is 'i'.

The following are all diadic numerical functions. They all deliver "illegalnum" if either of the inputs is a non-legal value (not 0 to 262143). In addition each function also has defined error conditions which will also deliver "illegalnum".

PLUS: (num: a b) -> bool
Numerical addition, an error occurs (result "illegalnum") if (a+b) > 262143, ie cannot be represented as a legal value.

MINUS: (num: a b) -> bool
Numerical subtraction (a - b), an error occurs if a < b, ie result should be negative.

TIMES: (num: a b) -> bool
Numerical multiplication, an error occurs if (a * b) > 262143, ie cannot be represented as a legal value.

C.2.3 List operations

CONS: (bool: a, boollist: b) -> boollist
This adds the value 'a' (as the most significant member) to a boollist 'b'. The function PASS is applied to 'a' before it is joined onto the boollist, so only the values 't', 'f' and 'i' can be added (all other values are turned to 'i's). If the boollist is full, that is consists of "maxlist" significant values (not 'bv'), then there is no room to add 'a' and a corrupted list ([maxlist]i) is delivered. Note specifically that adding anything to a corrupted list or an unknown list ([maxlist]x), results in a corrupted list.

HD: (boollist: a) -> bool
This delivers the most significant member of a boollist. Again, PASS is applied to the result, so only the values 't', 'f' and 'i' can be delivered. Specifically, applying HD to a empty boollist ([maxlist]bv) or a corrupt boollist ([maxlist]i) delivers 'i'.

TL: (boollist: a) -> boollist
This delivers the boollist formed by removing the most significant member from 'a'. Note that if 'a' is either an empty list, a corrupted list or an unknown list, the result is equal to a corrupted list.

NULL: (boollist: a) -> bool
This delivers 't' if 'a' is an empty list, 'i' if 'a' is a corrupted or an unknown list. Otherwise it delivers 'f'.

EL: (num: element, boollist: a) -> bool
This delivers the 'element' member of boollist 'a' (numbered from 0 to maxlist-1). The function PASS is applied to the result to ensure only 't', 'f' and 'i' can be delivered. The illegal value 'i' will be delivered if 'element' is greater than maxlist-1 or is not a legal number (ie not 0 to 262143). 'i' will also be delivered if 'a' is a corrupted list, or contains less than 'element'+1 significant values.

SEG: ([2]num: elements, boollist: a) -> boollist
This delivers the boollist formed by the members of 'a' from 'elements[1]' to 'elements[2]'. If the number of significant elements required (elements[2] - elements[1] + 1) is less than 1 or greater than maxlist, or if either 'elements[1]' or 'elements[2]' is not a legal number, the result is a corrupted list.

If the number of significant elements required in the result is legal (ie between 1 and maxlist), then the result will have that many significant elements (either 't', 'f' or 'i'). Specifically, applying SEG to a corrupted list, an unknown list, or an empty list will result in the appropriate number of 'i's.

The following case should be noted. If 'a' is a boollist, of potentially 32 members (ie maxlist = 32), with 10 significant (non-bv) values, and the operation SEG((number/7, number/12), a) is applied. The result is the boollist consisting of members 7 to 9 of 'a' together with three 'i's.

V: (boollist: a) -> num
This converts a boollist to the equivalent 'num'. If 'a' is a corrupted list, an empty list or an unknown list, the result will be 'illegalnum'. The result will also be 'illegalnum' unless all the significant members of 'a' are either 't' or 'f', or if the numerical equivalent of 'a' is greater than 262143.

C.3 Supported generic functions, implemented as ELLA MACRO's

The following functions exist as ELLA macro's. They are all involved with operations on 'word(n)'s, and are parameterised for (n). For example, the first function, WORD(n), exists in the library as a macro called WORDN. If the function WORD4 (ie num -> word4) is required the user must make the statement:-

"FN WORD4 = (num: a) -> word4: WORDN(4) a."

WORD(n): (num: a) -> word(n)
This converts a 'num' into its representation as a word(n). If 'a' is 'unknownnum' or 'illegalnum', or if 'a' cannot be represented in (n) bits (eg WORD4(number/16)) the result is "[n]i".

BITS(n): (word(n): a) -> boollist
This converts a word(n) into the equivalent boollist. The boollist will always have (n) significant members ('t', 'f' or 'i' only).

VAL(n): (word(n): a) -> num
This converts a word(n) into its appropriate numerical equivalent. It is identical to V(BITS(n) a).

NOT(n): (word(n): a) -> word(n)
This performs the NOT operation on each of the elements of 'a' to produce the result.

AND(n): (word(n): a b) -> word(n)
This performs the AND operation on each of the elements of 'a' and 'b' to produce the result (ie bit 1 of the result is a[1] AND b[1]).

OR(n): (word(n): a b) -> word(n)
This performs the OR operation on each of the elements of 'a' and 'b' to produce the result (ie bit 1 of the result is a[1] OR b[1]).

WORDEQU(n): (word(n): a b) -> bool
This function tests for equality between two 'word(n)'s. The definition of equality used is as follows: the equivalent bits from both 'word(n)'s are compared using EQUIV, if all (n) comparisons are 't' the 'word(n)'s are equal and 't' is delivered, if any of the comparisons is 'f' the 'word(n)'s cannot be equal, and so the result is 'f', any other result delivers 'i'.

Annex D: Description of the types and functions in the auxilary library

D.1 Supported types

D.1.1 result

This is an ELLA enumeration type used when comparing the values delivered by two functions, one of which claims to be an implementation of the other (the specification). It has three values:-

ok: The value delivered by the implementation function agrees with that required by the specification.

xxxbadspec: The value delivered by the specification function is illegal or bv (this should not occur).

xxxwrongxxx: The value delivered by the specification function is legal, but the implementation does not agree with it.

See section D.2.2 for the function based on this type.

D.1.2 latchmode

This is an ELLA enumeration type used in the specification of latch primitives for the 'bool' and 'word(n)' types. It has two values:-

verify: This indicates that the latch is to be used for verification. This means that the data and enable inputs to the latch directly affect the output (without delay), and that if the enable is 'f', then the output will be the "unaltered stored" value associated with the appropriate type (ie either 'q' for a bool latch or "[n]q" for a word(n) latch).

simulate: This indicates that the latch is to be used for simulation. This means that the data and enable inputs affect the latch outputs on the next simulation cycle (ie after one 'clock-tick'). Also, if the enable input is 'f', then the latch delivers the same value as during the last simulation cycle. That is it is acting as a true memory device.

See section 7 of the main body of this paper, and sections D.2.3 and D.3 for details of the latch functions.

D.1.3 testselect

This is an ELLA enumeration type which is used solely by the function DISPLAYRES. It has two values:-

all: This indicates that the results of all tests are to be printed.

failonly: This indicates that only those tests which result in errors are to be printed.

The use of DISPLAYRES and the testing strategy are more fully explained in the main body of this paper (section 6) and is used in the examples in Annexes G and H.

D.2 Supported simple functions-

D.2.1 Numerical operations

DIVIDE: (num: a b) -> bool
Integer division (a/b). The result is 'illegalnum' if either 'a' or 'b' is not a legal numerical value (ie not 0 to 262143) or if 'b' is zero.

REMAINDER: (num: a b) -> bool
Finds the remainder after performing the integer division (a/b), The result is 'illegalnum' if either 'a' or 'b' is not a legal numerical value (ie not 0 to 262143) or if 'b' is zero. Note that:-

PLUS(REMAINDER(a,b), TIMES(DIVIDE(a,b), b)) = a

For all legal 'a' and 'b', when 'b' not zero

TESTCOUNT: (bool: dummy) -> num
This function generates a sequence of numbers from zero to 262143. It is used when producing test vectors, to generate the current vector number. The first value generated is zero, so the ELLA simulator will perform test zero at time 0, etc. Should the simulator be run beyond time = 262143, the output of TESTCOUNT starts counting again from 0.

The 'bool' parameter to the function is not used, but must be present, as ELLA does not allow "(VOID) -> num" functions (to borrow ALGOL68 notation).

TESTWORD: (num: a) -> [18]bool
This function converts a 'num' to a row of 18 'bool's. It is used with TESTCOUNT to generate the current test number as a [18]bool. Its behaviour is similar to WORD18, but its operation is undefined if 'a' is not a legal value (TESTCOUNT cannot produce illegal values).

D.2.2 Compare functions

COMPBOOL: (bool: spec calc) -> result
This compares two bool values. One is the value delivered by a specification function (spec) and the other that calculated by an implementation function (calc). If 'spec' is 't', 'f', 'q', 'z' or 'oc' then if 'calc' has the same value, the specification function has been implemented correctly and the result is 'ok'. If 'calc' has some other value the result is 'xxxwrongxxx'. If 'spec' is 'x' then any value delivered by the implementation function is correct and gives the result 'ok'. No other value of 'spec' should be possible, so if any other value of 'spec' is found, the result is 'xxxbadspec'.

COMPNUM: (num: spec calc) -> result
This is a function analoguous to COMPBOOL for 'num's. If 'spec' is a legal value (0 to 262143) then 'calc' must have the same value to produce the result 'ok', otherwise the result is 'xxxwrongxxx'. If 'spec' is 'unknownnum', then any value of 'calc' gives the result 'ok'. Finally, 'spec' should not have the value 'illegalnum', so if it has, the result is 'xxxbadspec'.

COMPJOIN: (result: a b) -> result
It may be necessary to compare a number of seperate elements by use of COMPBOOL, COMPNUM and COMPWORD(n). COMPJOIN allows these seperate 'result's to be combined. Its function is analogous to AND between 'bool's.

If 'a' and 'b' are both 'ok' then the result is 'ok'. If either 'a' or 'b' is 'xxbadspec' then the result is 'xxxbadspec'. Otherwise the result is 'xxxwrongxxx'.

D.2.3 Latch function

LATCHBOOL: (bool: data enable, latchmode: mode) -> bool
This is the basic latch primitive for the storage of a single 'bool' value. It has two modes of operation depending whether the system is being verified or simulated ('mode' equals 'validate' or 'simulate'). The uses of these two modes is explained in section 7 of the main body of this paper.

During verification, if 'enable' is 't', then the output of LATCHBOOL is 'data' ('t', 'f', 'q' and 'x' only, all other inputs give 'i'). If 'enable' is 'f', then the output will be 'q', to indicate that the contents of the latch are not changed. Finally, if 'enable' has any other value, the output is indeterminate, 'i'.

During simulation, the states of 'enable' and 'data' affect the output on the next (not current) simulation cycle. If 'enable' is 't', then the next output of LATCHBOOL is 'data', provided 'data' is not 'q' ('t', 'f', 'x' or 'i' only, as described above). If 'enable' is 't' and 'data' is 'q', the value delivered on the next cycle will be the same as the current output. If 'enable' is 'f', then the next output will again be the same as the current output. Finally, if 'enable' has any other value, the next output is indeterminate, 'i'. During cycle 0 of the simulation the output is 'i', indicating that it is impossible to predict which state a latch will start in after switch-on.

It should be noted that the output in verification mode is normally the value that the simulation mode latch would produce on the next simulation cycle. The exception to this is that if the latch is disabled, 'enable' is 'f', the verification mode latch produces a special "marker" to indicate an unchanged value, whilst in simulation mode the actual value is known and hence delivered. Also using this "marker" value, 'q', as the data input has the same effect as disabling the latch.

REGBOOL: (bool: data enable, latchmode: mode) -> [2]bool
This function is like LATCHBOOL, but delivers a complementary pair of outputs. The first output is the same as would be delivered by LATCHBOOL, but the second is the inverse of the first if it is "t" or "f", but otherwise is the same as the first (ie "x" "q" or "i").

D.2.4 The Display functions

DISPLAYRES: (result: data, num: testnum, testselect: level) -> (num,result,bool)
DISPLAYBOOL: (bool: data control) -> bool

DISPLAYNUM: (num: data, bool: control) -> num
The use of these functions is explained in the main body of this paper, section 6.3.

D.2.5 Tri-state functions

BOOLTRI: (bool: data enable) -> bool
This function is used to generate tri-state circuit elements. If 'enable' is 't' then the output is "PASS data", (ie 't', 'f' or 'i'). However, if 'enable' is 'f' the output is 'z'. Any other value of 'enable' gives an output of 'i'.

JOINTRI: (bool: a b) -> bool:
This function models the connection of two tri-state signals. If 'a' and 'b' are both 'z' then the result is 'z'. If either is 'z' then the result is the PASS value of the other, but if neither 'a' nor 'b' is 'z' then the result is 'i'.

D.2.6 Open collector functions

OCOUTPUT: (bool: a) -> bool
This function is used to model the output of an open collector gate. If 'a' is 'f' it delivers 'f' (ie output enabled). If 'a' is 't' then it delivers 'oc' (ie the output is disabled). Under all other conditions the delivered value is 'i'.

WIREDOR: (bool: a b) -> bool
This function models the joining of two open collector outputs. If either 'a' or 'b' are 'f' then the delivered value is 'f'. If both 'a' and 'b' are 'oc' then the delivered value is 'oc', and in all other cases the value is 'i'.

PULLUP: (bool: a) -> bool
This function models a "pull-up" resistor applied to an open collector gate. That is it converts an open collector signal to a "normal" boolean signal. If 'a' is 'f' the result is 'f', if 'a' is 'oc' the result is 't'. Otherwise the result is 'i'.

D.3 Supported generic functions, implemented as ELLA MACRO's

The following functions exist as ELLA macro's. They are all involved with operations on 'word(n)'s, and are parameterised for (n). For example, the first function, REPWORD(n), exists in the library as a macro called REPWORDN. If the function REPWORD4 (ie word4 -> word4) is required the user must make the statement:-

```
"FN REPWORD4 = (word4: a) -> word4: REPWORDN(4) a."
```

REPWORD(n): (word(n): a) -> word(n)
This function overcomes a problem in ELLA created by the order in which elements are written in a denotation for a row of objects. For example, the LCF-LSM representation of the word4 with a decimal value of 11 is #1011. The same value expressed in ELLA is (t,t,f,t). Note that the bits are in the opposite order. As the aim of this library is to enable easy translation of LCF-LSM functions into ELLA, this bit reversal is regarded as unacceptable. However the function REPWORD(n) provides the bit reversal so that the LCF-LSM value #1011 is correctly generated by REPWORD4(t,f,t,t).

This function can also be used on word(n)'s delivered from a function, to ensure the ELLA simulator will print the delivered values in the expected order.

COMPWORD(n): (word(n): spec calc) -> result
This function compares two word(n)'s in a similar fashion to COMPBOOL (applied to two bool's). The result is 'ok' only if each bit of the calculated value (calc) agrees with the equivalent bit in the specification value (spec), using the algorithm stated in the description of COMPBOOL. If any of these bit tests, gives the result 'xxxbadspec', this is the delivered value from the function, otherwise the result is 'xxxwrongxxx'.

LATCHWORD(n): (word(n): data, bool: enable, latchmode: mode) -> word(n)
This is identical to (n) LATCHBOOL's, see section D.2.3.

REGWORD(n): (word(n): data, bool: enable, latchmode: mode) -> [2]word(n)
This is identical to (n) REGBOOL's, see section D.2.3.

WORD(n)TRI: (word(n): data, bool: enable) -> word(n)
This function is used to generate tri-state busses. If 'enable' is 't' then the output is 'data', (only 't', 'f' or 'i' values). However, if 'enable' is 'f' the output is "[n]z". Any other value of 'enable' gives an output of "[n]i" (cf BOOLTRI).

JOINTRI(n): (word(n): a b) -> word(n)
This function models the connection of two tri-state busses. It is identical to (n) uses of JOINTRI (ie output 1 is JOINTRI(a[1], b[1]) etc).

DISPLAYWORD(n): (word(n): data, bool: control) -> word(n)
The use of this function is explained in the main body of this paper, section 6.3 and D.2.4.

SELECT(n): (word(n): a, num: element) -> bool
This function is used to select an element from a row of 'bool's. Its effect is very similar to "EL(element, BITS(n) a)". However, whilst the above expression is capable of delivering "f", "t" or "i" values only, SELECT(n) can deliver any 'bool' value. It is used to select a particular element from a word(n) delivered from a specification or implementation function for comparison, for an example of use see Annex H.

OCWORD(n): (word(n): a) -> word(n)
This is equivalent to (n) OCOUTPUT's.

WIREDOR(n): (word(n): a b) -> word(n)
This is equivalent to (n) WIREDOR's.

PULLUP(n): (word(n): a) -> word(n)
This is equivalent to (n) PULLUP's.

Annex E: The LCF-LSM library in ELLA

```
INT maxlist = 32.          \ **** the maximum length for a boollist **** \
                           \ ****      18 < maxlist <= 256          **** \



\ ************************************************************************** \
\ ************************************************************************** \
\ *****                                                              ***** \
\ *****                    LOGICAL OPERATIONS                        ***** \
\ *****                                                              ***** \
\ ************************************************************************** \
\ ************************************************************************** \


TYPE  bool = NEW (t | f | i | x | q | bv | z | oc).
                                  \ true | false | dont-know | illegal \
                                  \ unaltered_stored_value |           \
                                  \ bv (used in boollist only) |       \
                                  \ high_impedance | open_collector    \

FN   NOT = (bool: a) -> bool:
        CASE a OF t:f, f:t ELSE i ESAC.


FN   AND = (bool: a b) -> bool:
        CASE (a,b) OF
             (t,t): t,  (f,bool): f ELSEOF (bool,f): f
          ELSE i
        ESAC.


FN    OR = (bool: a b) -> bool:
        CASE (a,b) OF
             (f,f): f,  (t,bool): t ELSEOF (bool,t): t
          ELSE i
        ESAC.


FN EQUIV = (bool: a b) -> bool:
        CASE (a,b) OF (t,t): t, (f,t): f, (t,f): f, (f,f): t ELSE i ESAC.


FN   XOR = (bool: a b) -> bool:
        CASE (a,b) OF (t,t): f, (f,t): t, (t,f): t, (f,f): f ELSE i ESAC.


FN  PASS = (bool: a) -> bool:
        CASE a OF f:f, t:t ELSE i ESAC.
```

```
\ ********************************************************************** \
\ ********************************************************************** \
\ *****                                                          ***** \
\ *****          - NUMERICAL OPERATIONS                          ***** \
\ *****                                                          ***** \
\ ********************************************************************** \
\ ********************************************************************** \

INT xxmaxword = 18.    \ **** Maximum length "wordn",  for VAX = 18 & **** \
                       \ **** 1900 = 14. If xxmaxword >= 20, logic of **** \
                       \ **** V and XXFINDBL must be changed          **** \

INT illegalnum = (1 SL xxmaxword),
    unknownnum = illegalnum + 1.


TYPE num = NEW number/(0..unknownnum).  \ 0 to (2**xxmaxword - 1) = legal \
                                        \   >= (2**xxmaxword) = illegal   \


FN EQUAL = (num: a b) -> bool:
   BEGIN   FN EQU = (num: a b) -> num:
              ARITH IF (a>=illegalnum) OR (b>=illegalnum)
                       THEN illegalnum
                       ELSE IF a = b THEN 1 ELSE 0 FI
                    FI.
           OUTPUT CASE EQU(a,b) OF
                       number/0: f, number/1: t
                    ELSE i
                  ESAC
     END.


FN  PLUS = (num: a b) -> num:
    ARITH IF (a + b) >= illegalnum THEN illegalnum ELSE (a + b) FI.


FN MINUS = (num: a b) -> num:
    ARITH IF (a < b) OR (a >= illegalnum)
                  THEN illegalnum
                  ELSE (a - b)
                FI.


FN TIMES= (num: a b) -> num:
    ARITH IF (a >= illegalnum) OR (b >= illegalnum) OR
             (((a SR 11) /= 0) AND ((b SR 11) /= 0))
          THEN illegalnum
          ELSE IF (a * b) >= illegalnum THEN illegalnum ELSE (a * b) FI
        FI.
```

```
\ ********************************************************************** \
\ ********************************************************************** \
\ *****                                                          ***** \
\ *****                    LIST OPERATIONS                       ***** \
\ *****                                                          ***** \
\ ********************************************************************** \
\ ********************************************************************** \


TYPE boollist = [maxlist]bool.          \ ****   empty list = all "bv" **** \
                                        \ **** corrupt list = all "i"  **** \


INT xxhalflist = IF maxlist <= 16
                   THEN IF maxlist <= 4
                          THEN IF maxlist <= 2 THEN 1 ELSE 2 FI
                          ELSE IF maxlist <= 8 THEN 4 ELSE 8 FI
                        FI
                   ELSE IF maxlist <= 64
                          THEN IF maxlist <=  32 THEN 16 ELSE  32 FI
                          ELSE IF maxlist <= 128 THEN 64 ELSE 128 FI
                        FI
                 FI.


MAC XXNORMMAC(INT n) = (boollist: a) -> boollist:
    IF n = 1
      THEN CASE a[maxlist] OF bv: (bv CONC (a[1..(maxlist-1)])) ELSE a ESAC
      ELSE XXNORMMAC(n SR 1)
             (CASE a[((maxlist+1)-n)..maxlist] OF
                   [n]bv: (([n]bv) CONC (a[1..(maxlist-n)]))
              ELSE a
              ESAC
             )
    FI.


FN XXNORM = (boollist: a) -> boollist: XXNORMMAC(xxhalflist) a.


MAC XXUNNORMMAC(INT n) = (boollist: a) -> boollist:
    IF n = 1
      THEN CASE a[1] OF bv: ((a[2..maxlist]) CONC bv) ELSE a ESAC
      ELSE XXUNNORMMAC(n SR 1)
             (CASE a[1..n] OF
                   [n]bv: ((a[(n+1)..maxlist]) CONC ([n]bv))
              ELSE a
              ESAC
             )
    FI.


FN XXUNNORM = (boollist: a) -> boollist: XXUNNORMMAC(xxhalflist) a.
```

```
\ ********************  CONS, HD, TL and NULL  ********************** \


FN CONS = (bool: a, boollist: b) -> boollist:
          CASE b[maxlist] OF
               bv: XXUNNORM(((XXNORM b)[2..maxlist]) CONC (PASS a))
            ELSE [maxlist]i
          ESAC.


FN   HD = (boollist: a) -> bool: PASS (XXNORM a)[maxlist].


FN   TL = (boollist: a) -> boollist:
          CASE a OF
               [maxlist]bv: [maxlist]i,
               [maxlist]x:  [maxlist]i,
               [maxlist]i:  [maxlist]i
            ELSE XXUNNORM(bv CONC ((XXNORM a)[1..(maxlist-1)]))
          ESAC.


FN NULL = (boollist: a) -> bool:
          CASE a OF
               [maxlist]bv: t,
               [maxlist]x:  i,
               [maxlist]i:  i
            ELSE f
          ESAC.




\ ************** macros/functions for EL and SEG ********************** \


FN XXVALIDEL = (num: a) -> num:
   ARITH IF a > (maxlist - 1) THEN illegalnum ELSE a FI.


FN XXNEEDED  = (num: level mask) -> num:
   ARITH IF level = mask
           THEN illegalnum          \last required shift      \
           ELSE level IAND mask     \shift required 0 or not 0\
         FI.


FN XXDIV2    = (num: mask) -> num: ARITH mask SR 1.


FN XXMASKBIT = (num: level mask) -> num: ARITH level IAND (INOT mask).
```

```
MAC XXSRNMAC(INT n) =  (boollist: a, num: level mask) -> boollist:
    BEGIN LET cn = XXNEEDED(level, mask).
      OUTPUT
        IF n = 1
          THEN ((a[2..maxlist]) CONC bv)
          ELSE CASE cn OF
                    number/illegalnum: ((a[(n+1)..maxlist]) CONC ([n]bv))
                 ELSE XXSRNMAC(n SR 1)
                        (CASE cn OF
                              number/0: a
                           ELSE ((a[(n+1)..maxlist]) CONC ([n]bv))
                         ESAC,
                         XXMASKBIT(level, mask), XXDIV2 mask
                        )
               ESAC
        FI
      END.



FN XXSRN = (num: level, boollist: b) -> boollist:
   CASE XXVALIDEL level OF
        number/illegalnum: [maxlist]i,
        number/0:          b
     ELSE XXSRNMAC(xxhalflist)(b, level, number/xxhalflist)
   ESAC.



FN XXFINDMASK = (num: from to) -> boollist:
   BEGIN FN XXSEGLENGTH = (num: from to) -> num:
            ARITH IF (to < from) OR (to >= illegalnum) OR
                     (((to - from) + 1 ) > maxlist)
                   THEN illegalnum
                   ELSE ((maxlist + from) - to - 1)
                 FI.
     OUTPUT XXSRN( XXSEGLENGTH(from,to), [maxlist]t)
    END.



\ ***************************  EL and SEG  *************************** \


FN   EL = (num: a, boollist: b) -> bool: PASS( (XXSRN(a, b))[1] ).



FN  SEG = ([2]num: segsel, boollist: b) -> boollist:
   BEGIN LET  moveright = XXSRN(segsel[1], b),
              findmask = XXFINDMASK(segsel[1], segsel[2]).
         OUTPUT  [INT k=1..maxlist]CASE findmask[k] OF
                                        t: PASS moveright[k]
                                     ELSE findmask[k]
                                   ESAC
    END.
```

```
\ ********************************** V ********************************** \


FN XXVM0 = (num: lsb msb shift) -> num:
   ARITH IF msb >= illegalnum
           THEN illegalnum
           ELSE IF ((msb SL shift) + lsb) >= illegalnum
                  THEN illegalnum
                  ELSE ((msb SL shift) + lsb)
                FI
         FI.


FN XXVM1 = (bool: a) -> num:
   CASE a OF
        f: number/0, t: number/1, bv: number/0
     ELSE number/illegalnum
   ESAC.


FN XXVM2 = ([2]bool: a) -> num:
           XXVM0(XXVM1 a[1], XXVM1 a[2], number/1).


FN XXVM4 = ([4]bool: a) -> num:
           XXVM0(XXVM2 a[1..2], XXVM2 a[3..4], number/2).


FN XXVM8 = ([8]bool: a) -> num:
           XXVM0(XXVM4 a[1..4], XXVM4 a[5..8], number/4).


FN XXVM20 = ([20]bool: a) -> num:
           XXVM0(XXVM0(XXVM8 a[1..8], XXVM8 a[9..16], number/8),
                 XXVM4 a[17..20],
                 number/16
                ).


FN V = (boollist: a) -> num:
   BEGIN LET paddedlist = a CONC [20]bv.
         OUTPUT  CASE a OF
                      [maxlist]bv: number/illegalnum
                   ELSE CASE paddedlist[21..(maxlist+20)] OF
                             [maxlist](f | bv): XXVM20(paddedlist[1..20])
                           ELSE number/illegalnum
                         ESAC
                 ESAC
     END.
```

```
\ ********** convert num to list of bools ********** \

FN XXLSB = (num: a bits) -> num: ARITH a IAND ((1 SL bits) - 1).
FN XXMSB = (num: a bits) -> num: ARITH (a SR bits) IAND ((1 SL bits) - 1).


FN XXFINDB2 = (num: a) -> [2]bool:
   CASE a OF
        number/0: (f,f), number/1: (t,f), number/2: (f,t), number/3: (t,t)
   ESAC.


FN XXFINDB4 = (num: a) -> [4]bool:
   XXFINDB2(XXLSB(a, number/2)) CONC XXFINDB2(XXMSB(a, number/2)).


FN XXFINDB8 = (num: a) -> [8]bool:
   XXFINDB4(XXLSB(a, number/4)) CONC XXFINDB4(XXMSB(a, number/4)).


FN XXFINDB16 = (num: a) -> [16]bool:
   XXFINDB8(XXLSB(a, number/8)) CONC XXFINDB8(XXMSB(a, number/8)).


FN XXFINDB32 = (num: a) -> [32]bool:
   XXFINDB16(XXLSB(a, number/16)) CONC XXFINDB16(XXMSB(a, number/16)).
```

```
\ ********************************************************************** \
\ ********************************************************************** \
\ *****                                                          ***** \
\ ***** MACRO FUNCTIONS, available for any value of (n)         ***** \
\ *****                                                          ***** \
\ *****         WORDn, VALn, BITSn, NOTn, ANDn, ORn              ***** \
\ *****                                                          ***** \
\ ********************************************************************** \
\ ********************************************************************** \


MAC WORDN(INT n) = (num: a) -> [n]bool:
    BEGIN LET rowconv = CASE a OF
                              number/illegalnum: [32]i,
                              number/unknownnum: [32]i
                          ELSE XXFINDB32 a
                        ESAC.
    OUTPUT CASE rowconv[(n+1)..32] OF [(32-n)]f: rowconv[1..n] ELSE [n]i ESAC
    END.


MAC BITSN(INT n) = ([n]bool: a) -> boollist:
    BEGIN LET amod = [INT k=1..n]PASS a[k].
          OUTPUT amod CONC ([maxlist-n]bv)
    END.


MAC  VALN(INT n) = ([n]bool: a) -> num:
    BEGIN LET amod = [INT k=1..n]PASS a[k].
          OUTPUT V(amod CONC ([maxlist-n]bv))
    END.


MAC  NOTN(INT n) = ([n]bool: a) -> [n]bool: [INT k=1..n]NOT a[k].


MAC  ANDN(INT n) = ([n]bool: a b) -> [n]bool: [INT k=1..n]AND(a[k], b[k]).


MAC   ORN(INT n) = ([n]bool: a b) -> [n]bool: [INT k=1..n] OR(a[k], b[k]).


MAC WORDEQUN(INT n) = ([n]bool: a b) -> bool:
    BEGIN LET equivrow = [INT k=1..n] EQUIV(a[k], b[k]).
          OUTPUT CASE equivrow OF
                      [n]t: t
                    ELSEOF [n](t | i): i
                    ELSE f
                 ESAC
      END.
```

Annex F: The auxilary library in ELLA

```
\ **** NUMERICAL OPERATIONS ********************************************** \


FN DIVIDE = (num: a b) -> num:
   ARITH IF (a >= illegalnum) OR (b >= illegalnum) OR (b = 0)
           THEN illegalnum
           ELSE a % b
         FI.


FN REMAINDER = (num: a b) -> num:
   ARITH IF (a >= illegalnum) OR (b >= illegalnum) OR (b = 0)
           THEN illegalnum
           ELSE  a - ((a % b) * b)
         FI.


FN TESTCOUNT = (bool: dummy) -> num:
   BEGIN FN NUMDEL = (num) -> num: DELAY(number/0, 1).
         FN INCNUM = (num: a) -> num:
                     ARITH IF a = (illegalnum - 1) THEN 0 ELSE a + 1 FI.
         MAKE NUMDEL: nd.
         JOIN INCNUM(nd) -> nd.
         OUTPUT nd
     END.


FN TESTWORD = (num: a) -> [xxmaxword]bool: (XXFINDB32 a)[1..xxmaxword].


MAC REPWORDN(INT n) = ([n]bool: a) -> [n]bool: [INT k=1..n] a[(n+1)-k].



\ **** COMPARISON OPERATIONS ********************************************* \

TYPE     result = NEW(  ok | xxxwrongxxx | xxxbadspec).

FN COMPBOOL = (bool: spec calc) -> result:
   CASE (spec, calc) OF
        (bv, bool): xxxbadspec,
        ( i, bool): xxxbadspec,
        ( x, bool): ok,
        (oc,   oc): ok,
        ( z,    z): ok,
        ( q,    q): ok,
        ( t,    t): ok,
        ( f,    f): ok
     ELSE xxxwrongxxx
   ESAC.
```

```
FN COMPNUM = (num: spec calc) -> result:
   CASE spec OF
        number/illegalnum: xxxbadspec,
        number/unknownnum: ok
     ELSE CASE EQUAL(spec, calc) OF t: ok ELSE xxxwrongxxx ESAC
   ESAC.


FN COMPJOIN = (result: a b) -> result:
   CASE (a,b) OF
        (ok,ok): ok,
        (xxxbadspec, result):  xxxbadspec
 ELSEOF (result, xxxbadspec):  xxxbadspec
     ELSE xxxwrongxxx
   ESAC.


MAC COMPWORDN(INT n) = ([n]bool: spec calc) -> result:
    BEGIN LET testrow = [INT k=1..n]COMPBOOL(spec[k], calc[k]).
          OUTPUT CASE testrow OF
                      [n]ok: ok
                  ELSEOF [n](ok | xxxwrongxxx): xxxwrongxxx
                  ELSE xxxbadspec
                ESAC
      END.


\ **** LATCH OPERATIONS ********************************************** \


TYPE  latchmode = NEW( verify | simulate).
TYPE testselect = NEW( all | failonly).


FN XXDELBOOL = (bool) -> bool: DELAY(1, 1).


FN LATCHBOOL = (bool: data enable, latchmode: mode) -> bool:
   BEGIN MAKE XXDELBOOL: delop.
         LET nochange = CASE mode OF verify: q, simulate: delop ESAC.
         LET  datasel = CASE enable OF
                              f: nochange,
                              t: CASE data OF
                                      t:t, f:f, x:x, q: nochange
                                   ELSE 1
                                 ESAC
                            ELSE 1
                          ESAC.
         JOIN datasel -> delop.
         OUTPUT CASE mode OF verify: datasel, simulate: delop ESAC
     END.


FN REGBOOL = (bool: data enable, latchmode: mode) -> [2]bool:
   BEGIN LET trueop = LATCHBOOL(data, enable, mode).
         OUTPUT (trueop, CASE trueop OF t:f, f:t ELSE trueop ESAC)
     END
```

```
MAC LATCHWORDN(INT n) = ([n]bool: data, bool: enable, latchmode: mode) ->
                        [n]bool:
                        [INT k=1..n] LATCHBOOL(data[k], enable, mode).


MAC REGWORDN(INT n) = ([n]bool: data, bool: enable, latchmode: mode) ->
                      [2][n]bool:
    BEGIN LET  oprow = [INT k=1..n] REGBOOL(data[k], enable, mode).
          LET trueop = [INT k=1..n] oprow[k][1].
          LET  invop = [INT k=1..n] oprow[k][2].
          OUTPUT (trueop, invop)
     END.



\ **** DISPLAY OPERATIONS ********************************************** \

FN DISPLAYRES = (result: data, num: testnum, testselect: level) ->
                (num, result, bool):
   BEGIN FN DELRES = (num,result) -> (num,result): DELAY((number/0,ok), 1).
         MAKE DELRES: delop.
         LET  op = CASE level OF
                        all: (testnum, data, t),
                   failonly: CASE data OF
                                  ok: (delop[1], delop[2], f)
                               ELSE (testnum, data, t)
                             ESAC
                   ESAC.
         JOIN (op[1],op[2]) -> delop.
         OUTPUT op
    END.


FN DISPLAYBOOL = (bool: data control) -> bool:
   BEGIN MAKE XXDELBOOL: delop.
         LET  datasel = CASE control OF f: delop, t: data ELSE i ESAC.
         JOIN datasel -> delop.
         OUTPUT datasel
    END.


FN DISPLAYNUM = (num: data, bool: control) -> num:
   BEGIN FN DELNUM = (num) -> num: DELAY(number/illegalnum, 1).
         MAKE DELNUM: delop.
         LET  datasel = CASE control OF
                             f: delop, t: data
                          ELSE number/illegalnum
                        ESAC.
         JOIN datasel -> delop.
         OUTPUT datasel
    END.
```

```
MAC DISPLAYWORDN(INT n) = ([n]bool: data, bool: control) -> [n]bool:
   BEGIN FN DELWN = ([n]bool) -> [n]bool: DELAY([n]i, 1).
         MAKE DELWN: delop.
         LET  datasel = CASE control OF f: delop, t: data ELSE [n]i ESAC.
         JOIN datasel -> delop.
         OUTPUT datasel
     END.


MAC SELECTN(INT n) = ([n]bool: row, num: element) -> bool:
    (XXSRN(element, row CONC ([maxlist-n]i)))[1].



\ **** TRISTATE OPERATIONS ********************************************** \


FN BOOLTRI = (bool: data enable) -> bool:
   CASE enable OF  f: z,  t: PASS data ELSE i ESAC.


FN JOINTRI = (bool: a b) -> bool:
   CASE (a,b) OF (z,z): z, (z,t): t, (z,f): f, (t,z): t, (f,z): f ELSE i ESAC.


MAC  WORDNTRI(INT n) = ([n]bool: data, bool: enable) -> [n]bool:
   CASE enable OF  f: [n]z,  t: [INT k=1..n]PASS data[k]  ELSE [n]i ESAC.


MAC JOINTRIN(INT n) = ([n]bool: a b) -> [n]bool:
    [INT k=1..n] JOINTRI(a[k], b[k]).



\ **** OPEN COLLECTOR OPERATIONS ***************************************** \

FN OCOUTPUT = (bool: a) -> bool: CASE a OF f: f,  t: oc  ELSE i ESAC.


FN  WIREDOR = (bool: a b) -> bool:
    CASE (a,b) OF (oc, oc): oc,  (f, bool): f ELSEOF (bool,f): f  ELSE i ESAC.


FN PULLUP = (bool: a) -> bool: CASE a OF f: f,  oc: t  ELSE i ESAC.


MAC  OCWORDN(INT n) = ([n]bool: a) -> [n]bool: [INT k=1..n]OCOUTPUT a[k].

MAC WIREDORN(INT n) = ([n]bool: a b) -> [n]bool: [INT k=1..n]WIREDOR(a[k],b[k]).

MAC  PULLUPN(INT n) = ([n]bool: a) -> [n]bool: [INT k=1..n]PULLUP a[k].
```

Annex G: An example of verification of a combinatorial circuit

G.1 The specification

The example to be verified is based on the four input multiplexer described in the main body of the paper (see section 3). However, to show the effect of coversion from LCF-LSM to ELLA, the specification has been changed slightly, so the selector (which was two 'bool's called select0 and select1) is now a single word2.

In LCF-LSM the specification for the required multiplexer is:-

```
SPEC :(bool#bool#bool#bool#word2) -> bool
SPEC  (   a,   b,   c,   d, selector) ==
LET selectval = VAL2 selector IN     (numerical equivalent of selector)
   (selectval = 0 -> a |
    selectval = 1 -> b |
    selectval = 2 -> c |
                     d               (by implication selectval = 3)
   )
```

This can be converted into ELLA as follows:-

```
TYPE word2 = [2]bool.

FN VAL2 = (word2: a) -> num: VALN(2) a.

FN SPEC = (bool: a b c d, word2: selector) -> bool:
   BEGIN LET selectval = VAL2 selector.
  OUTPUT CASE selectval OF
              number/0: PASS a,
              number/1: PASS b,
              number/2: PASS c,
              number/3: PASS d
            ELSE i
         ESAC
     END.
```

Notice the explicit declaration of the type "word2" and the function "VAL2" created by the macro function "VALN". Other than the lexical differences, the two main changes between the LCF-LSM version and that in ELLA are the need to apply PASS to each of the delivered values (for the reason explained in section 5 of the main body of the paper) and the need for an explicit "ELSE" limb to the case clause, as if "selectval" is not number/0, number/1 or number/2 it cannot be assumed to be number/3 as it was in LCF-LSM (because of the possiblities of illegal or indeterminate values).

G.2 The implementation

The implementation is the same as described in the main body of the paper (see section 3 and Fig 1). Note that the functionality of the basic logic elements is described in terms of the LCF-LSM library functions detailed in Annex C.

```
FN   INV = (bool: a)       -> bool: NOT a.
FN NAND3 = (bool: a b c)   -> bool: NOT(a AND b AND c).
FN NAND4 = (bool: a b c d) -> bool: NOT(a AND b AND c AND d).

FN CIRCUIT = (bool: a b c d, word2: selector) -> bool:
   BEGIN LET sel0bar = INV selector[1],
             sel1bar = INV selector[2],
             sel0buf = INV sel0bar,
             sel1buf = INV sel1bar.
      OUTPUT NAND4(NAND3(a, sel0bar, sel1bar),
                   NAND3(b, sel0buf, sel1bar),
                   NAND3(c, sel0bar, sel1buf),
                   NAND3(d, sel0buf, sel1buf)
                  )
      END.
```

G.3 The test vectors

As explained in section 3 of the main body of the paper, eight tests are required to prove correspondance between the specification and implementation of the multiplexer. The function to generate these tests is shown below. As with the function TESTCOUNT (see D.2.1), the function TESTVECTORS requires a dummy input parameter, as "VOID -> ...." is not an allowed function type in ELLA.

The value "testnum", is a 'num' with values in the range 0 to 7. It is used as the test number, and is the first member of the structure delivered from TESTVECTORS. The value "testrow" is a structure of type (bool,bool,bool,bool,word2) and is the required input for both SPEC and CIRCUIT. The value of "testrow" is selected by "testnum". Note that tests 0 and 1 test the first input of the multiplexer (with the other three marked as irrelevant, "x"), tests 2 and 3 the second input, etc. Also notice the use of REPWORD2 (see D.3) to create a 'word2' from a pair of 'bool's.

```
FN TESTVECTORS = (bool: dummy) -> (num, (bool,bool,bool,bool,word2)):
   BEGIN FN REPWORD2 = (word2: a) -> word2: REPWORDN(2) a.
         LET testnum = REMAINDER( TESTCOUNT dummy, number/8),
             testrow = CASE testnum OF
                              number/0: (f, x, x, x, REPWORD2(f,f)),
                              number/1: (t, x, x, x, REPWORD2(f,f)),
                              number/2: (x, f, x, x, REPWORD2(f,t)),
                              number/3: (x, t, x, x, REPWORD2(f,t)),
                              number/4: (x, x, f, x, REPWORD2(t,f)),
                              number/5: (x, x, t, x, REPWORD2(t,f)),
                              number/6: (x, x, x, f, REPWORD2(t,t)),
                              number/7: (x, x, x, t, REPWORD2(t,t))
                       ESAC.
          OUTPUT (testnum, testrow)
      END.
```

G.4 The comparison function

As SPEC and CIRCUIT deliver a single 'bool', the function COMPARE (as described in section 6.2 of the main paper) can be achieved simply by use of COMPBOOL.

```
FN COMPARE = (bool: spec circuit) -> result: COMPBOOL(spec, circuit).
```

G.5 The display function

It was decided that in addition to the test number and the result of the comparison, the values delivered by SPEC and CIRCUIT would be printed. The function DISPLAY is therefore as shown below, as described in section 6.3 of the main paper.

```
FN DISPLAY = (testselect: mode, result: data, num: testnum,
              bool: spec circuit)
          -> (num, bool, bool, result)
   BEGIN LET dr = DISPLAYRES(data, testnum, mode).
      OUTPUT (dr[1],
              DISPLAYBOOL(spec, dr[3]),
              DISPLAYBOOL(circuit, dr[3]),
              dr[2]
             )
    END.
```

G.6 The complete test program

The five functions, described above, are "joined together" into a single function, with the test vector from TESTVECTORS applied to SPEC and CIRCUIT, the results from these being compared, and the result from DISPLAY being delivered to the simulator (cf Fig 2). Note that this function has a single input parameter (to be controlled by the simulator), which is the 'testselect' mode required by the DISPLAY function.

```
FN RUNTESTS = (testselect: mode) -> (num, bool, bool, result):
   BEGIN LET testvectors = TESTVECTORS x,
                    spec = SPEC    testvectors[2],
                 circuit = CIRCUIT testvectors[2],
                 compare = COMPARE(spec, circuit).
         OUTPUT DISPLAY(mode, compare, testvectors[1], spec, circuit)
    END.
```

The results of running these functions is shown below (for mode = all) proving that the circuit is a valid implementation of the specification.

```
RUNTESTS = number/0 f f ok   (test-number, spec, circuit, result)
RUNTESTS = number/1 t t ok
RUNTESTS = number/2 f f ok
RUNTESTS = number/3 t t ok
RUNTESTS = number/4 f f ok
RUNTESTS = number/5 t t ok
RUNTESTS = number/6 f f ok
RUNTESTS = number/7 t t ok
```

Annex H: An example of verification of a circuit containing memory

H.1 The requirement

The example to be described is based on the specification of a six-bit counter. Informally, the counter is to have four modes of operation, determined by a pair of input signals called "func". The effects are to be as follows:-

```
func = 0 :  The current contents of the counter, "count" remains unchanged
func = 1 :  counter loaded from a six-bit input bus, called "loadin"
func = 2 :  count := count + 1
func = 3 :  count := count + 2
```

In addition to this requirement, there is also to be a signal from the counter which indicates when the content of the counter is 63 (ie all bits true), see Fig 6 in main body of paper. This can be expressed formally in LCF-LSM as:-

```
NEXTSTATE :(word6#word6#word2) -> word6     (find next state of count)
NEXTSTATE  (count,loadin,func) ==
   LET funcnum = VAL2 func IN
   LET   value = VAL6 count IN
   (funcnum = 0 -> count  |
    funcnum = 1 -> loadin |
    funcnum = 2 -> (value = 63 -> WORD6 0 | WORD6(value + 1) ) |
                   (value = 62 -> WORD6 0 |
                    value = 63 -> WORD6 1 |
                                  WORD6(value + 2)
                   )
   )


COUNT63 :(word6) -> bool            (indicate when current state = 63)
COUNT63   (count) ==
   (count = #111111)


SPEC :(word6#word6#word2) -> (word6#bool)   (combine above functions)
SPEC  (count,loadin,func) ==
   ((NEXTSTATE count loadin func), (COUNT63 count))
```

Note that when addition is performed, the effect of the finite word length has to be considered. So in the limb of NEXTSTATE that finds the next value of 'count' when "funcnum = 2", the special case of the current state of the counter being 63 is explicity handled (as 64 cannot be represented by a six-bit value), while for all other initial states of the counter the result is "value + 1". Also notice that the specification of COUNT63 could have been written as "((VAL6 count) = 63)", and that this representation is identical to the function as written.

H.2 The ELLA specification

The LCF-LSM specification presented above can be translated into the following ELLA form:-

```
TYPE word2 = [2]bool,
     word6 = [6]bool.

FN  VAL2 = (word2: a) -> num:   VALN(2) a.
FN  VAL6 = (word6: a) -> num:   VALN(6) a.
FN WORD6 = (num:   a) -> word6: WORDN(6) a.


FN STATE_LATCH = (word6: data, latchmode: mode) -> word6:
                 LATCHWORDN(6)(data, t, mode).


FN NEXTSTATE =(word6: count loadin, word2: func, latchmode: mode) -> word6:
   BEGIN LET   funcnum = VAL2 func.
         LET     value = VAL6 count.
         LET nextstate = CASE funcnum OF
                                 number/0: [6]q,
                                 number/1: loadin,
                                 number/2: CASE value OF
                                                 number/63: WORD6 number/0
                                             ELSE WORD6(value PLUS number/1)
                                           ESAC,
                                 number/3: CASE value OF
                                                 number/63: WORD6 number/1,
                                                 number/62: WORD6 number/0
                                             ELSE WORD6(value PLUS number/2)
                                           ESAC
                             ELSE [6]i
                           ESAC.
         OUTPUT STATE_LATCH(nextstate, mode)
     END.


FN COUNT63 = (word6: count) -> bool:  WORDEQUN(6)( count, (t,t,t,t,t,t) ).


FN SPEC = ((word6,word6,word2): signals, latchmode: mode) -> (word6,bool):
   BEGIN LET  count = signals[1].
         LET loadin = signals[2].
         LET   func = signals[3].
         OUTPUT (NEXTSTATE(count, loadin, func, mode), COUNT63 count)
     END.
```

There are a number of subtleties in the translation from LCF-LSM to ELLA that were not covered in the combinatorial example in Annex G, that need to be expanded.

i) The LCF-LSM description ignores the difference between the COUNT63 output (which is valid for the current state of the system) and the NEXTSTATE output (which is the next state of 'count'). In the ELLA version the function STATE_LATCH is introduced to indicate that NEXTSTATE defines the next state of the 'count', ie binds the memory shown in Fig 6.b to the NEXTSTATE function. The function STATE_LATCH is generated from the generic function LATCHWORD(n). This means that there is a requirement for a 'latchmode' input to NEXT_STATE (not present in the LCF-LSM version). Note that, as NEXT_STATE defines the next state ('word6' value) of 'count' under all circumstances, the 'enable' input of the LATCHWORD6 in STATE_LATCH can be fixed as 't'.

ii) The values delivered by the LCF-LSM version of NEXTSTATE and its ELLA equivalent, when "funcnum = 0", should be compared. In the LCF-LSM version the delivered value is "count", indicating that the count does not change. In the ELLA version, this is indicated by a delivered value of "[6]q".

iii) Note that in the description of the requirement, H.1, it was said that the function COUNT63 could be specified (with identical effects) as either:-

count = #111111 or (VAL6 count) = 63

The ELLA translations of these have slightly different interpretations. In the second case, when the requirement is expressed in terms of comparisons between 'num's, if any of the bits of 'count' are illegal values (not 't' or 'f') the result is indeterminate (which is not allowed in a specification), whilst in the first case provided all the bits of 'count' are 't', or at least one is 'f', the result of the comparison is determinate. This means that when the implementation of this part of the requirement is tested (see H.6) only seven tests are needed, whilst if the second form of the expression had been used, all 64 legal values of (VAL6 count) would need to be examined.

iv) In the final specification function SPEC, a 'latchmode' value is required, to be passed on to NEXTSTATE. In order to simplify the testing functions RUNTESTS (see H.4, H.5 and H.6), the inputs to SPEC that will have corresponding values in the implementation function, CIRCUIT, are grouped into a single structure of type "(word6,word6,word2)", the 'latchmode' input to SPEC is a separate formal parameter.

H.3 The implementation

The circuit that has been produced to meet the above requirement is shown in Fig H.1. The description of this diagram in ELLA follows. Note that as the circuit description will only ever be used for verification, the latch function, LATCH, doesn't need a 'latchmode' input, but can be permanently in verify mode, and that a function MPLXBIT is used to describe a circuit function that is repeated many times.

```
FN   INV = (bool: a)        -> bool: NOT a.
FN  NOR2 = (bool: a b)      -> bool: NOT(a  OR b).
FN NAND2 = (bool: a b)      -> bool: NOT(a AND b).
FN NAND3 = (bool: a b c)    -> bool: NOT(a AND b AND c).
FN NAND4 = (bool: a b c d) -> bool: NOT(a AND b AND c AND d).
FN  XNOR = (bool: a b)      -> bool: a EQUIV b.

FN LATCH = (word6: data, bool: enable) -> word6:
              LATCHWORDN(6)(data, enable, verify).


FN CIRCUIT = (word6: count loadin, word2: func) -> (word6,bool):
   BEGIN FN MPLXBIT = (bool: countn cbn loadn selload selcount) -> bool:
            NAND2(NAND2 (selload, loadn),
                  NAND2 (selcount, XNOR(countn, cbn))
                 ).

         LET  selload = INV func[2],
             selcount = INV selload,
             notfunc0 = INV func[1],
             func0buf = INV notfunc0.

         LET c0 = MPLXBIT(count[1], func0buf, loadin[1], selload, selcount).

         LET carry1b = NOR2(count[1], func0buf).
         LET  carry1 = INV carry1b.

         LET c1 = MPLXBIT(count[2], carry1b, loadin[2], selload, selcount).
         LET c2 = MPLXBIT(count[3], NAND2(carry1, count[2]),
                          loadin[3], selload, selcount
                         ).
         LET c3 = MPLXBIT(count[4], NAND3(carry1, count[2], count[3]),
                          loadin[4], selload, selcount
                         ).

          LET carry4b = NAND4(carry1, count[2], count[3], count[4]).
          LET carry5b = NAND2(INV carry4b, count[5]).

          LET c4 = MPLXBIT(count[5], carry4b, loadin[5], selload, selcount).
          LET c5 = MPLXBIT(count[6], carry5b, loadin[6], selload, selcount).

          LET count63 = NOR2( NAND2(count[1], count[6]), carry5b).

          OUTPUT (LATCH((c0, c1, c2, c3, c4, c5), NAND2(selload, notfunc0)),
                  count63
                 )
      END.
```

H.4 Verifying the "count" output, "func" equals 0, 2 or 3

Verification of the above circuit can be divided into two main parts. The first of these is to show that the next value of 'count' will be correct, the second, to show the value indicating 'count' equals 63 is also correct.

The first of these is further subdivided, depending upon whether the next value of 'count' needs to be treated as a single 'word6', or as six separate bits. This section will describe the tests on 'count' treated as a single 'word6'.

When arithmetic operations are involved, it must be shown that all possible values give the correct result. That is for 'func' equal to 2 or 3 all 64 possible value of 'count' must be tried ('loadin' being irrelevant, ie "[6]x"). A single test is also required for 'func' equal to 0, to show both SPEC and CIRCUIT give "[6]q". The following functions provide the appropriate test vectors and perform the tests, in the same style as described in Annex G. Note that the test vectors 0 to 63 test 'func' = 2 ('count' = 0 to 63), vectors 64 to 127 test 'func' = 3 ('count' = 0 to 63), and vector 128 tests 'func' = 0.

```
FN TESTVECTORS = (bool: dummy) -> (num, (word6,word6,word2)):
   BEGIN LET    testnum = REMAINDER(TESTCOUNT dummy, number/129),
                testrow = TESTWORD testnum,
             testvector = CASE testrow[7..8] OF
                                (f,f): (testrow[1..6], [6]x, (f,t)),
                                (t,f): (testrow[1..6], [6]x, (t,t))
                            ELSE ([6]x, [6]x, (f,f))
                          ESAC.
         OUTPUT (testnum, testvector)
     END.


FN COMPARE = (word6: spec circuit) -> result: COMPWORDN(6)(spec, circuit).


FN DISPLAY = (testselect: mode, result: data, num: testnum,
              word6: spec circuit) -> (num, word6, word6, result):
   BEGIN FN DISPLAYWORD6 = (word6: w6data, bool: control) -> word6:
                           DISPLAYWORDN(6)(w6data, control).
         LET dr = DISPLAYRES(data, testnum, mode).
         OUTPUT (dr[1], DISPLAYWORD6(spec,    dr[3]),
                        DISPLAYWORD6(circuit, dr[3]),
                 dr[2]
                )
     END.


FN RUNTESTS = (testselect: mode) -> (num, word6, word6, result):
   BEGIN LET testvectors = TESTVECTORS x.
         LET        spec = (    SPEC(testvectors[2], verify) )[1].
         LET     circuit = ( CIRCUIT testvectors[2]          )[1].
         LET     compare = COMPARE(spec, circuit).
         OUTPUT DISPLAY(mode, compare, testvectors[1], spec, circuit)
     END.
```

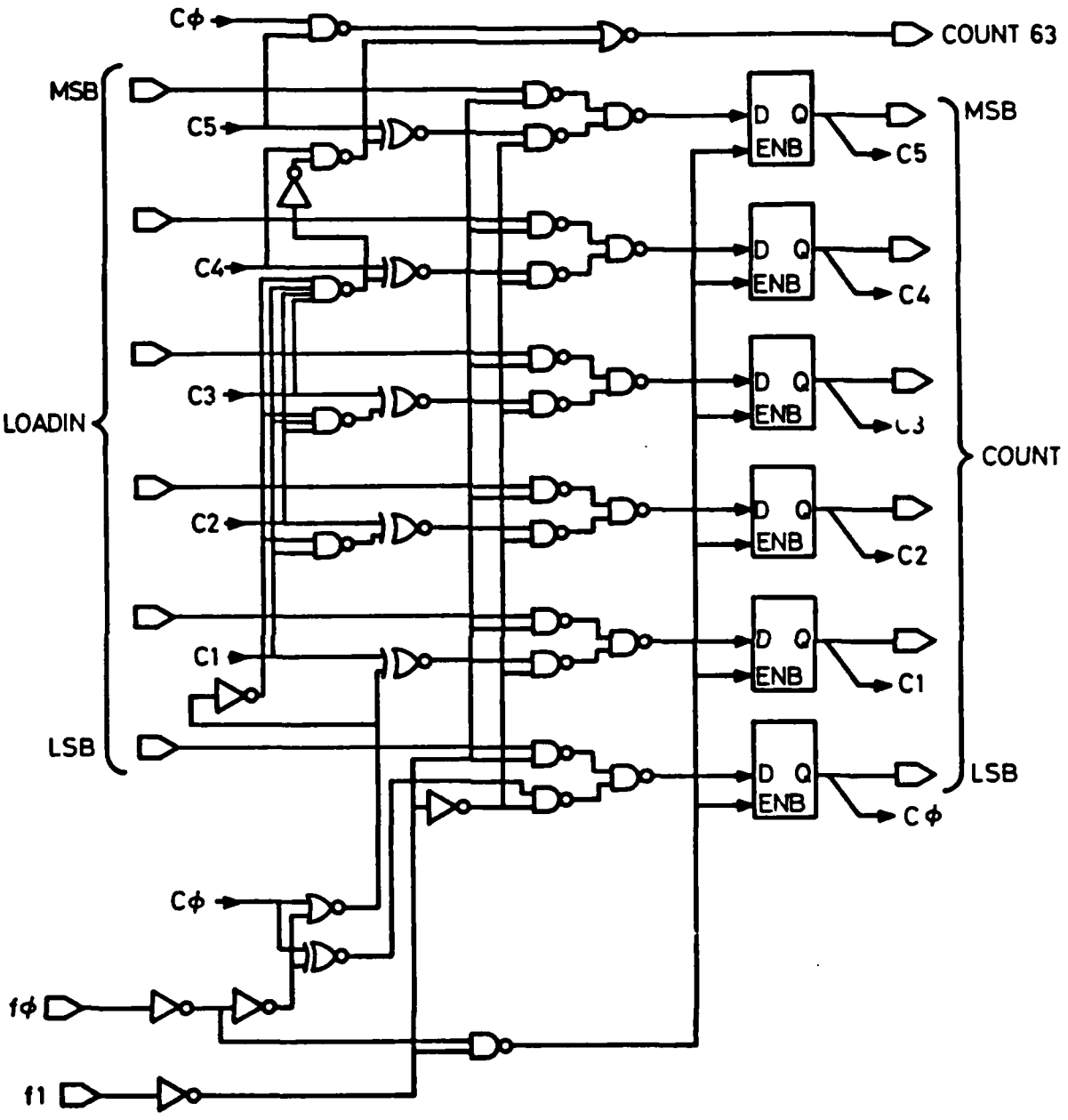


FIG. H.1 COUNTER CIRCUIT

DOCUMENT CONTROL SHEET

Overall security classification of sheet ..... Unclassified .....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference	3. Agency Reference	4. Report Security Classification
	Report 85012		

5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location
	Royal Signals & Radar Establishment

5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location

7. Title Formal proof of correspondence between the specification of a hardware module and its gate level implementation.

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials	9(a) Author 2	9(b) Authors 3,4...	10. Date	pp. ref.
PYGOTT C H				

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement

Unlimited

Descriptors (or keywords)

continue on separate piece of paper

Abstract

The growing use of digital circuits in safety critical environments and the cost of correcting mistakes in large scale integrated circuits, both lead to a requirement for a high level of confidence in the correctness of the design of micro-electronic elements.

This paper describes a novel application of a general hardware description language that enables the implementation of a synchronous circuit to be checked exhaustively against a high level, implementation independent, specification of its functionality (originally written in a formalism such as first order predicate calculus). The technique avoids the cost, in

/simulation

DOCUMENT CONTOL SHEET (contd)

simulation time, usually associated with exhaustive checking.

The method is illustrated by examples written in the design and description language ELLA: no prior knowledge of ELLA is assumed. Included in the annexes to this paper are a library of ELLA functions that provide those facilities required for the validation of circuits, and the translation of specifications written in the first order predicate calculus language LCF-LSM into ELLA.

DTIC

END

4-86